VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Algoritmy pro řešení rozvrhovacích problémů s omezenými zdroji

Algorithms for Resource Constraint Project Scheduling Problem

2017 Bc. Viliam Frolo

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Viliam Frolo**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Algoritmy pro řešení rozvrhovacích problémů s omezenými zdroji
Algorithms for Resource Constraint Project Scheduling Problem

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je implementovat několik algoritmů pro řešení problému označovaného jako RCPSP - resource constraint project scheduling problem.
V tomto problému jde o to, že máme zadáno n činností, které je třeba provést, přičemž u každé z těchto činností je určena doba trvání. Navíc jsou mezi jednotlivými činnostmi určeny závislosti určující, že některé činnosti se mohou začít provádět až poté, co jiné byly dokončeny. Dále je zadáno m zdrojů (např. stroje, na kterých se činnost provádí, nebo pracovníci,vykonávající danou činnost, apod.), přičemž každý zdroj má dánu určitou omezenou kapacitu. Pro jednotlivé činnosti je pak dáno, kolik z kapacita daného zdroje je pro provedení dané činnosti potřeba. Cílem je najít takové rozvržení daných činností, při kterém jsou dodrženy všechny omezující podmínky (tj. jsou dodrženy závislosti mezi činnostmi a není překročena kapacita žádného zdroje), při kterém je celková doba provedení všech činností co nejmenší.

1. Nastudujte algoritmy popsané v článku B. Franck, K. Neumann, C. Schwindt (2001).
2. Implementujte tyto algoritmy.
3. Otestuje jednotlivé algoritmy na vhodně zvolených testovacích datech a porovnejte je z hlediska jejich efektivity.

Seznam doporučené odborné literatury:

[1] B. Franck, K. Neumann, C. Schwindt - Truncated branch-and-bound, schedule-construction, and schedule-improvement procedures for resource-constrained project scheduling, OR Spektrum 23, pp. 297-324, Springer-Verlag, 2001.

Další literatura podle pokynů vedoucího práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Zdeněk Sawa, Ph.D.**

Datum zadání: 01.09.2015

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

V Ostrave 28. apríla 2017 .....

Súhlasím so zverejnením tejto diplomovej práce podľa požiadaviek čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostrave 28. apríla 2017 .....

Touto cestou sa chcem úprimne poďakovať svojmu školiteľovi doc. Ing. Zdeňku Sawovi, Ph.D., za jeho cenné rady, odborné usmernenia, pomoc a podporu. Moje poďakovanie chcem venovať aj mojej manželke a celej rodine za podporu, ktorú mi poskytli pri písaní diplomovej práce.

Abstrakt

Cieľom diplomovej práce je implementácia niekoľkých algoritmov pre riešenie problémov známych ako RCPSP – resource constraint project scheduling problem. Takýto problém môžeme popísať ako súbor zdrojov s obmedzenou kapacitou, ktoré majú byť priradené k súboru vykonávaných aktivít alebo úloh. Aktivity majú definované časové trvanie a musia byť vykonávané v danom poradí. Dané algoritmy budú vytvárať rozvrhy aktivít projektu. Budeme sa zameriavať na rozvrhy, pre ktoré sú dodržané všetky závislosti medzi aktivitami, a nie je prekročená kapacita žiadneho zdroja. Popíšeme si jednotlivé algoritmy riešiace tento problém a dátové štruktúry, ktoré budeme pri riešení problému používať. Zameriame sa na dva typy algoritmov – algoritmus vetiev a hraníc a jeho skrátené verzie a algoritmy pracujúce s prioritnými pravidlami. Nakoniec prevedieme sériu testov, ktoré nám vyhodnotia efektívnosť implementovaných algoritmov.

Kľúčové slová: riešenie rozvrhovacích problémov s obmedzenými zdrojmi, skrátené metódy vetiev a hraníc, metódy prioritných pravidiel, projekt, rozvrh

Abstract

The aim of the diploma thesis is the implementation of several algorithms for solving problems known as RCPSP - resource constraint project scheduling problem. This problem can be described as a set of resources with limited capacity to be associated with a set of activities or tasks. Activities have a defined time duration and must be performed in given order. The algorithms create schedules for project activities. We will focus on schedules for which all dependencies between activities are respected and no capacity of any resource is exceeded. We will describe the algorithms that solve this problem and the data structures we will use. We focus on two types of algorithms - branch and bound algorithm and its truncated varsions and algorithms working with priority rules. Finally, we will perform a series of tests to evaluate the effectiveness of implemented algorithms.

Key Words: resource-constrained project scheduling, truncated branch-and-bound, priority rule methods, project, schedule

Obsah

Zoznam použitých skratiek a symbolov

RCPSP	–	Problém plánovania projektu s využívaním zdrojov, z angl. „*Resource constraint project scheduling problem*“
C#	–	Programovací jazyk C sharp
VH	–	Vetva a hranica
PP	–	Prioritné pravidlo
UB	–	Horná hranica z angl. „*Upper bound*“
LB	–	Dolná hranica z angl. „*Lover bound*“
LBD	–	Deštruktívna dolná hranica z angl. „*Destructive lower bound*“
LBW	–	Dolná hranica na základe vyťaženosti zdrojov z angl. „*Lower bound workload*“
S	–	Rozvrh z angl. „*Schedule*“
ES	–	Rozvrh najskorších začiatkov z angl. „*Erliest start*“
LS	–	Rozvrh najneskorčích začiatkov z angl. „*Latest start*“
δ	–	Časové oneskorenie
p	–	Doba trvania aktivity
d^{min}	–	Minimálne časové oneskorenie
d^{max}	–	Maximálne časové oneskorenie
C	–	Cyklus
$\mathcal{A}$	–	Aktívna množina
LST	–	Najmenší najneskorší začiatok z angl. „*Latest start time*“
MST	–	minimálny voľný čas z angl. „*Minumum slack time*“
MTS	–	najviac celkových nasledovníkov z angl. „*Most total succesors*“
LPF	–	najdlhšia nasledujúca cesta z angl. „*Longest path following*“
RSM	–	metóda plánovania zdrojov z angl. „*Resource scheduling method*“

Zoznam obrázkov

Zoznam tabuliek

1 Úvod

Problém plánovania činností sa rieši každý deň, avšak nie vždy je daný plán vyhotovený optimálne. To má za následok predĺženie celkového riešenia. Stretávame sa s ním napríklad pri rozvrhovaní pracovných zmien alebo pri procese výroby.

V tejto práci sa budeme zameriavať na riešenie problému plánovania projektu s obmedzenými zdrojmi (RCPSP). Tento problém môžeme definovať ako súbor zdrojov s obmedzenou kapacitou, ktoré majú byť priradené k súboru vykonávaných aktivít alebo úloh. Tieto aktivity sú vzájomne prepojené pomocou časových oneskorení, čo zaručí, že dané aktivity budú vykonávané v stanovenom poradí. Dá sa povedať, že ide o vytvorenie plánu aktivít, ktoré majú dané časové trvanie a využívajú určité zdroje. Aktivity musia byť vykonané v určitom poradí a musia tiež dodržať využitie potrebných zdrojov. Každá aktivita musí byť vykonaná až do konca, bez prerušenia. Pre každý problém existuje veľké množstvo možností, ako jednotlivé aktivity rozvrhnúť, preto je dôležité každú možnosť ohodnotiť. To znamená, že cieľom je nájsť vhodné priradenie zdrojov k aktivitám tak, aby boli splnené všetky definované obmedzenia a vhodne optimalizovať toto riešenie. Pri tvorbe plánu sa najčastejšie kladie dôraz na tieto dve hodnoty:

Minimálna doba trvania - ide o častejšie požadovaný cieľ v projektovom plánovaní. Ide o to, aby bol projekt dokončený za čo najkratšiu dobu, inými slovami ide o celkovú dobu od začiatku až po koniec projektu.

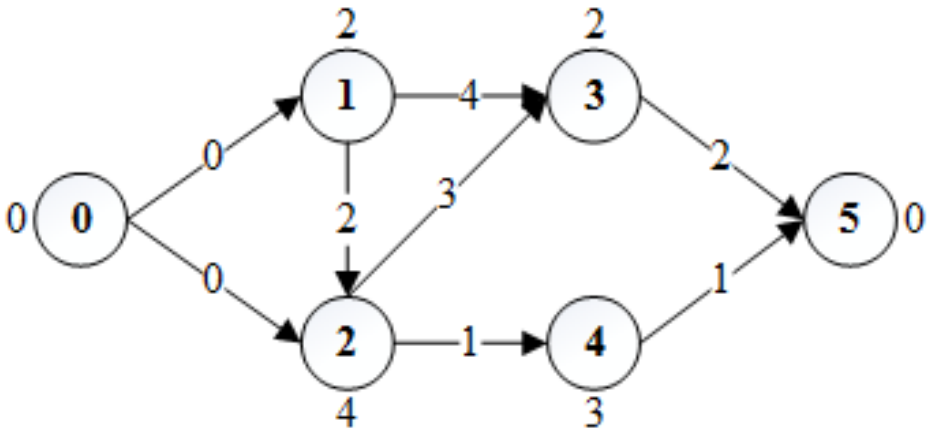
Minimalizovanie nákladov - spočíva v znížení nákladov na využívaných zdrojoch. Ide o efektívne využite takého množstva zdrojov v danom čase, ako je to možné.

V práci sa budeme zameriavať na riešenie problémov pomocou heuristických algoritmov: *Metóda vetiev a hraníc* a *Metóda prioritných pravidiel.*

2 Definícia pojmov

V tejto časti si vysvetlíme základné pojmy, ktoré sa týkajú rozhodovacích problémov projektu s obmedzenými zdrojmi.

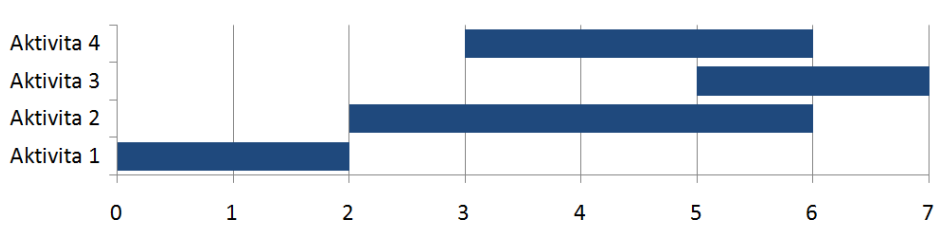
Projekt si môžeme predstaviť ako súbor reálnych aktivít, pričom jednotlivé aktivity označujeme celým číslom *1,...,n*. Každý projekt musí obsahovať začínajúcu aktivitu *0* a koncovú aktivitu *n+1*, pričom začínajúca aktivita nemá žiadnu predchádzajúcu aktivitu a z koncovej aktivity nevedie cesta do žiadnej nasledujúcej aktivity. Preto môžeme projektovú množinu aktivít označiť ako *V={0,1,...,n+1}*. Štruktúra tohto problému môže byť zobrazená ako acyklický graf, kde uzly predstavujú aktivity, oblúky predstavujú prednostné vzťahy a hodnota pri vrchole ukazuje dobu trvania aktivity. Takýto graf projektu máme znázornený na obrázku 1, ktorý je tvorený štyrmi reálnymi aktivitami.



Obr. 1: Projekt znázornený ako orientovaný graf

Plán alebo rozvrh projektu, ktorý označujeme S, uchováva informáciu o tom, kedy bude daná aktivita začínať, pričom začiatočná aktivita vždy začína v čase nula $S_0=0$. Každá aktivita i má definované trvanie alebo dobu spracovania p_i, pričom je daná ako celé nezáporné číslo. Taktiež má definovaných svojich nasledovníkov alebo nasledujúce aktivity, ktoré sa nemôžu začať vykonávať skôr, ako táto aktivita. Medzi každou prepojenou dvojicou aktivít existuje časové oneskorenie, ktoré sa nazýva aj váha medzi aktivitami. Je definované ako celé číslo, pričom môže ísť o minimálne alebo maximálne oneskorenie. Minimálne časové oneskorenie medzi aktivitou i a nasledujúcou aktivitou j, $(i, j \in V)$, označované ako d_{ij}^{min} znamená, že začiatok aktivity j môže byť zrealizovaný najskôr až po uplynutí tejto doby od začiatku aktivity i, pričom nemôže ísť o rovnakú aktivitu $i \neq j$ a táto hodnota musí byť minimálne nulová alebo väčšia, $d_{ij}^{min} \geq 0$ a S_j- $S_i \geq d_{ij}^{min}$. Maximálne časové oneskorenie, označované ako d_{ij}^{max}, medzi aktivitou i a jej nasledovníkom j, $(i, j \in V)$, pričom opäť platí $i \neq j$, určuje, že aktivita j musí

začať najneskôr do tejto doby od začiatku aktivity i, $S_j - S_i \leq d_{ij}^{max}$, pričom $d_{ij}^{max} \leq 0$. Plán, ktorý spĺňa všetky tieto obmedzenia, patrí do množiny časovo splniteľných plánov, označovanej ako $\mathcal{S}_T$. Na obrázku 2 máme graficky znázornený rozvrh projektu, pričom jeho definíciou boli



Obr. 2: Časový rozvrh projektu

hodnoty z obrázku 1. To znamená, že projekt tvoria štyri reálne aktivity. Osy rozvrhu označujú čas t, čo môže byť ľubovoľná časová jednotka a názvy aktivít.

Už zo zadania je zrejmé, že náš projekt bude definovaný aj zdrojmi. Preto budeme potrebovať množinu $\mathcal{R}$ zdrojov, ktoré sú potrebné na realizáciu aktivít projektu. Každý zdroj k, pričom $k \in \mathcal{R}$, má definovanú celkovú kapacitu $R_k > 0$, čo môžeme chápať ako dostupné množstvo daného zdroja. Existujú dva druhy zdrojov, a to obnoviteľné zdroje a neobnoviteľné zdroje. Neobnoviteľným zdrojom môže byť napríklad pracovný materiál, čo znamená, že ho môžeme využívať ľubovoľne počas celej doby vykonávania projektu, avšak nesmie byť prekročená jeho kapacita. V našom zadaní sme sa však nezaoberali neobnoviteľnými zdrojmi, pretože pokiaľ nie je prekročená ich celková kapacita, tak sa nám žiadnym spôsobom neovplyvní rozvrh projektu. Najdôležitejšiu úlohu v procese plánovania projektu zohrávajú obnoviteľné zdroje. Príkladom obnoviteľného zdroja môže byť napríklad pracovník, či stroj, pričom by sa mali tieto zdroje používať čo najefektívnejšie. To je dôležité z toho dôvodu, že v mnohých prípadoch sa jedná o zdroje, ktoré sú časovo spoplatnené. Takéto obnoviteľné zdroje sú po vykonaní aktivity opäť uvoľnené a môžu byť znovu použité pri vykonávaní inej aktivity. Zároveň však musíme opäť dodržať pravidlo, ktoré hovorí, že v žiadnom časovom úseku nemôže byť prekročená celková kapacita daného zdroja. Každá aktivita i má určené, koľko z daného zdroja potrebuje r_{ik} na to, aby mohla byť vykonaná, pričom $0 \leq r_{ik} \leq R_k$ $i \in V, k \in \mathcal{R}$. Pokiaľ náš plán spĺňa všetky obmedzenia na zdroje, ide o takzvaný zdrojovo splniteľný plán, ktorý spadá do množiny označovanej $\mathcal{S}_R$.

Všetky splniteľné plány sa nachádzajú v prieniku časovo splniteľných plánov a zdrojovo splniteľných plánov, $\mathcal{S} = \mathcal{S}_T \bigcap \mathcal{S}_R$. Ak sa na prepojenie projektových aktivít pozeráme ako na orientovaný graf (obrázok 1), môžeme určiť silne súvislé komponenty grafu. Silne súvislý komponent je taký maximálny podgraf orientovaného grafu, v ktorom pre každú dvojicu vrcholov u a v existuje sled. Každý komponent, ktorý sa skladá minimálne z dvoch aktivít nazývame cyklus C plánu. Každý takýto cyklus môžeme nazvať podprojektom, a z toho vyplýva nasledujúci teorém:

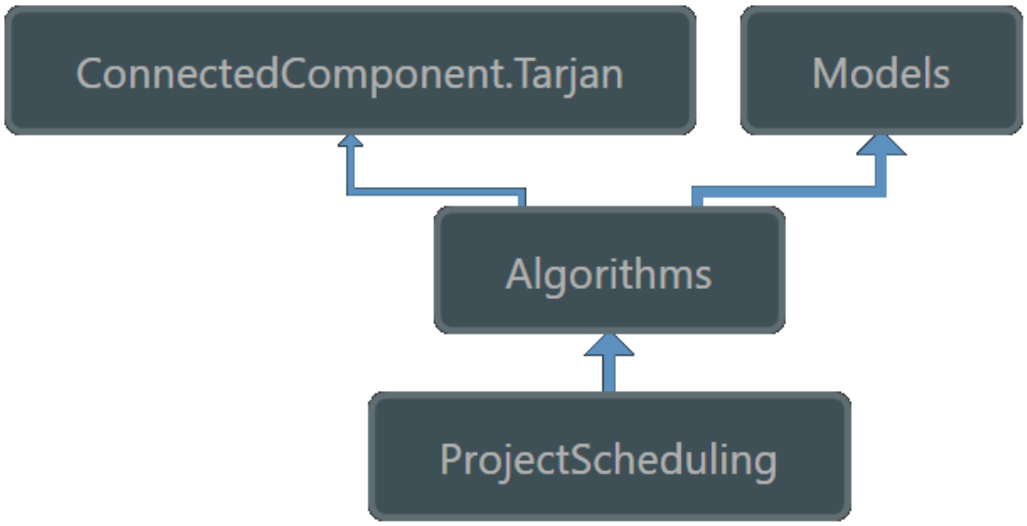
Teorém 1 *Pre projekt existuje uskutočniteľný plán práve vtedy a len vtedy, ak pre každý cyklus projektu existuje uskutočniteľný podplán.*

V procese plánovania sa budeme stretávať aj s nasledujúcimi pojmami:

- **UB** - horná hranica trvania projektu
- **LB** - dolná hranica trvania projektu
- **ES** - rozvrh najskorších možných štartov aktivít
- **LS** - rozvrh najneskorších možných štartov aktivít

3 Rozdelenie riešenia

Pre implementáciu riešenia problému RCPS sme zvolili programovací jazyk C#. Celkové riešenie je rozdelené do viacerých knižníc tried a program sa spúšťa v podobe konzolovej aplikácie. Takéto rozdelenie do viacerých projektov sme zvolili z toho dôvodu, aby bol náš zdrojový kód prehľadný, prípadne aby bolo možné využiť jednotlivé knižnice aj v prípade iných riešení, ako napríklad pri použití iného používateľského prostredia. V tejto kapitole si popíšeme jednotlivé knižnice tried, k čomu ich budeme využívať a aké obsahujú triedy. Jednotlivé implementácie a využitie vlastností a metód si popíšeme v ďalších kapitolách.



Obr. 3: Rozdelenie projektov riešenia a ich vzájomná závislosť

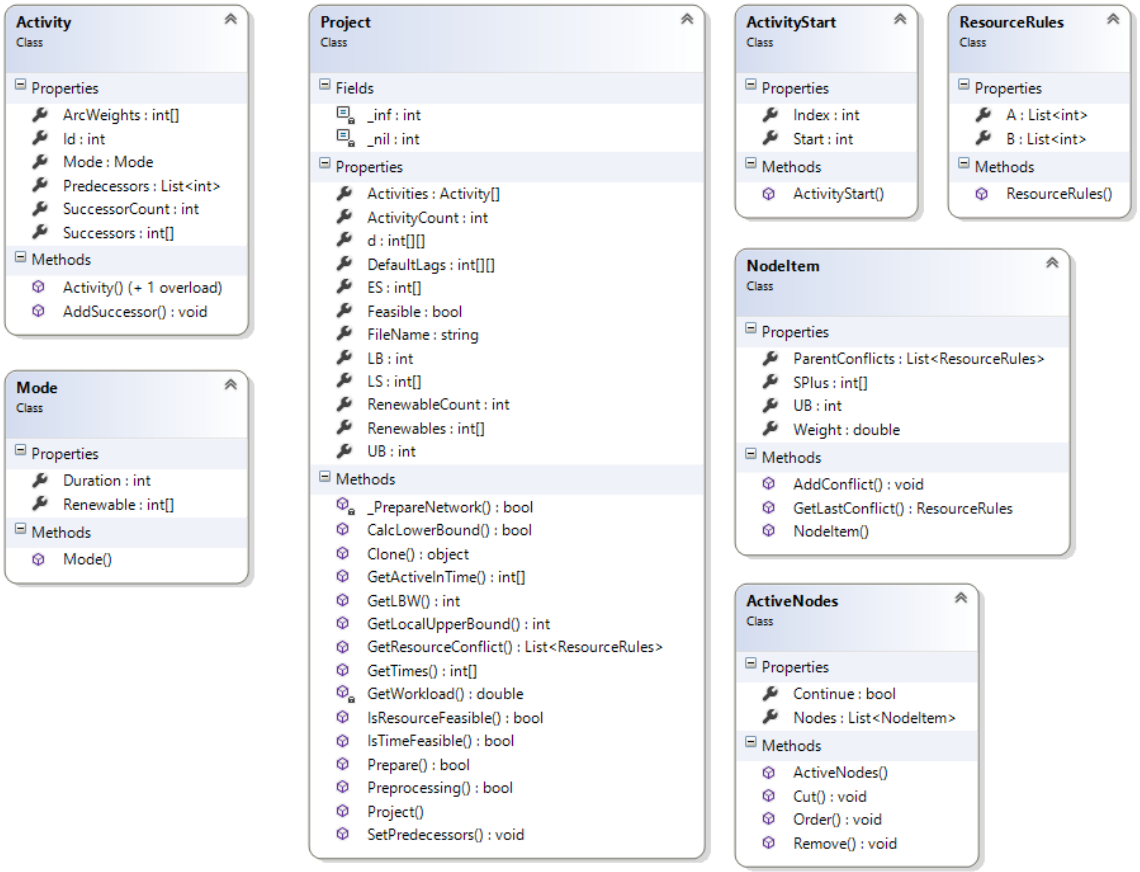
3.1 Tarjanov algoritmus

Ako sme už uviedli v kapitole 2 ,budeme sa pri našich riešeniach stretávať so situáciou, kedy budeme potrebovať určiť všetky cykly C daného projektu. Na to budeme potrebovať algoritmus, ktorý nám vyhľadá všetky silné komponenty z grafu projektu. Pre ich určenie sme zvolili Tarjanov algoritmus [2], ktorý vychádza z prehľadávania do hĺbky, pričom vrcholy si indexujeme podľa poradia ich navštívenia. Pri návrate z rekurzie sa každému vrcholu priradí uzol s najnižším indexom, na ktorý môže dosiahnuť. Všetky vrcholy, ktoré majú totožný cieľový uzol (index), sú v rovnakom komponente. Celkovú implementáciu tohto typu algoritmu možno nájsť v projekte *ConnectedComponent.Tarjan*. V našej práci však implementáciu tohto typu popisovať nebudeme, nakoľko nie priamo súvsí s naším zadaním. Existuje viacero iných algoritmov, poprípade je možné použiť už existujúce knižnice, ktoré riešia tento problém. Pre využívanie našej

implementácie je potrebné definovať graf , ktorý bude pre nás určený zoznamom vrcholov typu *Vertex*, pričom každý z nich bude znázorňovať jednu našu aktivitu projektu. Trieda *Vertex* obsahuje identifikátor aktivity, jej nasledovníkov a najskorší možný začiatok. Takto vytvorený graf pošleme do metódy *DetectCycle*, implementovanej v triede *ConnectedComponentFinder*, ktorá nám vráti zoznam všetkých silných komponentov grafu, pričom každý komponent má v sebe zoznam aktivít, z ktorých sa skladá. Keď budeme v našej práci spomínať určovanie cyklov, budeme na to využívať túto knižnicu.

3.2 Dátové štruktúry

Dôležitou úlohou v programovaní je aj správny návrh a implementácia dátových štruktúr, potrebných pre riešenie daného problému. Teraz si popíšeme knižnicu tried *Models*, ktorá obsahuje dátové štruktúry, ktoré budeme využívať pri hľadaní riešenia nášho problému. Predstavíme si ich vlastnosti a metódy, ktoré si podrobne popíšeme v nasledujúcich kapitolách. Všetky triedy, ich vlastnosti a dátové typy, metódy a ich návratovú hodnotu môžeme vidieť na triednom diagrame 4.



Obr. 4: Diagram tried

3.2.1 Mode

Každá aktivita projektu má stanovenú dobu trvania a počet zdrojov potrebných na jej vykonanie. Tieto informácie tvoria takzvaný mód aktivity, pričom existujú projekty, ktoré môžu mať pre jednu aktivitu viacero variánt vykonania, módov. V našom prípade sa však budeme zaoberať projektami s jedným módom pre každú aktivitu.

Vlastnosti:

- Duration - doba spracovania aktivity, trvanie
- Renewable - zoznam obnoviteľných zdrojov pre danú aktivitu

3.2.2 Activity

Táto trieda bude uchovávať informácie o aktivite.

Vlastnosti:

- Id - identifikátor aktivity, ktorý je zároveň aj ukazovateľom pozície aktivity v projekte
- ArcWeights - zoznam váh medzi aktivitou a nasledovníkmi, pričom poradie zodpovedá poradiu v zozname nasledovníkov
- Predecessors - zoznam predchodcov
- SuccessorCount - počet nasledovníkov
- Successors - zoznam nasledovníkov
- Mode - mód aktivity

Metódy:

- AddSuccessor - pridanie nového nasledovníka danej aktivite

3.2.3 Project

V tejto triede budeme mať všetky informácie týkajúce sa riešeného projektu.

Vlastnosti:

- ActivityCount - počet aktivít tvoriacich projekt
- Activities - zoznam všetkých aktivít projektu

- d - sieť uchovávajúca časové závislosti medzi aktivitami
- DefaultLags - sieť uchovávajúca pôvodné časové závislosti medzi aktivitami
- ES - zoznam najskorších začiatkov aktivít
- Feasible - informácia o tom, či je projekt splniteľný
- FileName - názov vstupného súboru
- LB - dolná hranica trvania
- LS - zoznam najneskorších začiatkov aktivít
- RenewablesCout - počet obnoviteľných zdrojov
- Renewables - zoznam kapacity obnoviteľných zdrojov
- UB - horná hranica trvania

Metódy:

- CalcLowerBound - vypočíta a nastaví LB
- GetActiveInTime - vráti zoznam aktívnych aktivít
- GetLocalUpperBound - vypočíta a vráti lokálnu hornú hranicu
- GetResourceConflict - vráti zoznam rozdelených aktivít podľa obmedzení na zdroje
- GetTimes - vráti zoznam časov pre daný rozvrh
- IsResourceFeasible - overí, či rozvrh spĺňa obmedzenia na zdroje
- IsTimeFeasible - overí, či rozvrh spĺňa časové oneskorenia
- Prepare - doplní hodnoty do siete d
- Preprocessing - vykoná fázu predspracovania
- GetLBW - vráti dolnú hranicu trvania podľa využitia zdrojov
- GetWorkload - výpočet pracovnej záťaže
- _PrepareNetwork - pripraví sieť d

3.2.4 ResourceRules

Trieda znázorňujúca pravidlo zdrojov. Definíciu tohoto pravidla si predstavíme v kapitole 5.1.

Vlastnosti:

- A - zoznam identifikátorov aktivít, ktoré môžu bežať súčasne
- B - zoznam identifikátorov čakajúcich aktivít

3.2.5 NodeItem

Znázorňuje jeden vrchol v stromovej štruktúre, ktorú budú používať algoritmy typu vetiev a hraníc.

Vlastnosti:

- ParentConflicts - zoznam pravidiel zdrojov zo všetkých predchádzajúcich vrcholov
- SPlus - zoznam začiatkov aktivít
- UB - horná hranica rozvrhu
- Weight - ohodnotenie vrcholu

Metódy:

- AddConflict - pridanie zdrojového pravidla do zoznamu pravidiel
- GetLastConflict - vráti posledne pridané pravidlo zo zoznamu

3.2.6 ActiveNodes

Označuje štruktúru stromu obsahujúci nespracované vrcholy, ktorú budeme používať pri algoritmoch vetiev a hraníc.

Vlastnosti:

- Continue - vráti informáciu o tom, či existuje v zozname ešte nejaký vrchol
- Nodes - zoznam vrcholov

Metódy:

- Cut - odstránenie vrcholov podľa hornej hranice
- Order - zoradí vrcholy podľa váhy
- Remove - odstráni prvý záznam zo zoznamu vrcholov

3.2.7 ActivityStart

Jednoduchá štruktúra, ktorá bude uchovávať len informáciu o začiatočnom čase v rozvrhu pre danú aktivitu a identifikátor tejto aktivity. Túto triedu budeme využívať pri určovaní aktívnej množiny.

Vlastnosti:

- Index - identifikátor aktivity
- Start - začiatočný čas aktivity

3.3 Algoritmy riešenia

Implementáciu všetkých algoritmov, ktoré budeme používať na hľadanie riešenia, obsahuje knižnica *Algorithms*, ktorej triedy sme si rozdelili nasledovne.

3.3.1 Base

Je základnou abstraktnou triedou, z ktorej budú dediť všetky ostatné triedy, pričom bude nutné implementovať abstraktnú metódu *Calculate*, ktorá bude obsahovať samotný algoritmus riešenia. Jej implementáciu si vždy podrobne popíšeme.

Vlastnosti:

- Feasible - udržiava informáciu o tom, či bolo nájdené riešenie
- BestUb - doposiaľ najlepšia nájdená horná hranica
- BestScheduleSoFar - doposiaľ najlepšie nájdené riešenie
- Project - zadanie projektu
- LastIndex - ukazovateľ na poslednú aktivitu projektu

Metódy:

- Calculate - abstraktná metóda, ktorú budeme implementovať pre každý typ algoritmu

3.3.2 DirectMethod

Algoritmus metódy *Calculate* si popíšeme v kapitole 6.3. Pri riešení bude využívať len vlastnosti triedy *Base* 3.3.1

3.3.3 DecompositionMethod

Postup riešenia, ako aj vysvetlenie všetkých pojmov spojených s jej riešením si vysvetlíme v kapitole 5.4.

Vlastnosti:

- RemoveBackward - táto vlastnosť nám bude slúžiť na to, či chceme odstrániť spiatočné oblúky

3.3.4 PriorityRules

V tejto triede budeme implementovať jednotlivé prioritné pravidlá a algoritmus hľadania riešenia s ich použitím.

Vlastnosti:

- LS - rozvrh najneskorších začiatkov aktivít pre aktuálne riešenie
- ES - rozvrh najskorších začiatkov aktivít pre aktuálne riešenie
- u - počet prevedení „odplánovania“
- UMax - maximálny počet „odplánovaní“
- rK - profily zdrojov
- Cycles - všetky nájdené cykly projektu
- C - zoznam spracovaných aktivít
- CMin - zoznam nespracovaných aktivít
- NodesCountToFinish - pre každú aktivitu udržiava informáciu, koľko aktivít bude vykonaných za ňou až po poslednú
- RuleFunc - metóda prioritného pravidla, ktorú používame

3.3.5 BnBBase

Rozšírenie základnej triedy o vlastnosti „stromu“, z ktorej budú dediť všetky ostatné triedy, obsahujúce algoritmus vetiev a hraníc - *Enumeration, Approximation, FilteredBeamSearch, DecompositionApproach*.

Vlastnosti:

- Tree - strom nespracovaných rozvrhov typu *ActiveNodes*

- Lambda - hodnota určujúca dôraz na obmedzenie

Metódy:

- GetNodeWeight - určenie váhy vrcholu stromu
- SetResourceLags - zavedenie obmedzení do rozvrhu
- GetPermutations - zistenie všetkých permutácií prvkov bez opakovania

3.3.6 Enumeration

Táto trieda obsahuje pôvodný postup hľadania riešenia metódou vetiev a hraníc, kapitola 5.1.

3.3.7 Approximation

Obsahuje implementáciu skráteného algoritmu vetiev a hraníc ε-aproximačná metóda, ktorej algoritmus si popíšeme v kapitole 5.2.

Vlastnosti:

- Epsilone - hodnota presnosti ε
- OriginalUB - pôvodná horná hranica projektu

3.3.8 FilteredBeamSearch

V tejto triede sa nachádza algoritmus vyhľadávania filtrovaním lúčov, ktorý si popíšeme v kapitole 5.3.

Vlastnosti:

- FilterWidth - šírka filtra
- MaxLeaves - maximálny počet listov
- FirstOnly - má sa spracovať len prvé pravidlo zdrojov

3.3.9 DecompositionApproach

Trieda obsahujúca algoritmy založené na riešení problému metódou rozkladu na menšie problémy, kapitola5.4.

Vlastnosti:

- Epsilone - hodnota presnosti ε

- MaxLeaves - maximálny počet listov
- RemoveBackward - táto vlastnosť nám bude slúžiť na to, či chceme odstrániť spiatočné oblúky

4 Určenie základných konceptov

V tejto kapitole si popíšeme metódy určovania základných konceptov projektu a ukážeme si spôsoby overovania splniteľnosti rozvrhu.

4.1 Dočasný projektový rozvrh

Ako už vieme z definície projektu, medzi aktivitami existujú časové oneskorenia δ, ktoré nazývame aj váha medzi aktivitami. Tieto časové oneskorenia sú dodržané ak platí, že ak odpočítame začiatok aktivity i od nasledujúcej aktivity j, bude tento rozdiel väčší alebo rovný váhe medzi nimi

$$S_j - S_i \geq \delta_{ij} \tag{1}$$

Na základe týchto časových oneskorení, dokážeme určiť rozvrhy *ES* a *LS* a hornú hranicu trvania projektu *UB*, ktorých postup si teraz popíšeme. Ako sme už spomínali, projekt si môžeme predstaviť ako orientovaný graf (obrázok 1), pričom jednotlivé vrcholy sú aktivity projektu. Pre takýto graf dokážeme určiť najdlhšie možné cesty medzi jednotlivými vrcholmi, ktoré budeme potrebovať na určenie spomínaných veličín. Na uchovanie týchto ciest použijeme naše dvojrozmerné pole *d*, kde prvý index bude označovať danú aktivitu a druhý aktivitu, do ktorej vedie cesta. Najskôr si pripravíme toto pole o veľkosti n, kde n je počet všetkých aktivít projektu, vrátane začiatočnej a koncovej aktivity. Pomocou cyklu s indexom i a vnoreným cyklom s indexom j, kde oba majú veľkosť n, pričom i a j si môžeme predstaviť ako vrcholy v grafe a začneme nastavovať hodnoty nášho poľa nasledovne. Ak sú indexy cyklov rôzne, nastavíme na túto pozíciu v poli hodnotu mínus nekonečno, $d_{ij} := -\infty$, pričom táto hodnota bude pre nás určovať to, že sa nedokážeme dostať z aktivity i do aktivity j. V prípade, že sa naše indexy rovnajú, nastavíme hodnotu na 0, $d_{ij} := 0$, pretože aktivita sa dostane sama do seba za čas nula. Do takto predpripraveného poľa vložíme reálne hodnoty. Začneme prechádzať všetky projektové aktivity i a pre každú prejdeme všetkých jej nasledovníkov s a nastavíme váhu medzi nimi, $d_{is} := -\delta_{is}$. V prípade, že by mala aktivita nastavenú váhu sama na seba, nemá zmysel pokračovať, pretože takto zadaný projekt nemá riešenie. Takto popísanú prípravu siete najdlhších ciest máme implementovanú v metóde *__PrepareNetwork*, ktorá nám vráti informáciu o tom, či sme zadanie zamietli, alebo nie. Ak sme nezamietli zadanie projektu, dokážeme určiť hornú hranicu trvania projektu *UB*. V niektorých prípadoch sa môžeme stretnúť s tým, že projekt má stanovenú hornú hranicu, ktorú by nemal prekročiť, avšak v našom prípade je potrebné si ju dopočítať. Vypočítame ju ako súčet z maxima medzi dobou trvania p_i aktivity i a maximálnej váhy všetkých jej nasledovníkov s, $UB := \sum_{i \in V} \max(p_i, \max_{<i,j> \in \delta_{ij}})$. Takto vypočítaná horná hranica sa nazýva aj lokálna horná hranica a budeme ju označovať ako $\overline{d}$. Na získanie lokálnej hornej hranice sme si vytvorili metódu *GetLocalUpperBound*. Taktiež z takejto siete d dokážeme určiť predchodcov aktivity. Opäť budeme prechádzať všetky aktivity i a pre ňu všetky aktivity j, ak $i \neq j$ a cesta medzi nimi je väčšia alebo rovná nule, a zároveň cesta z j do i je menšia ako nula, vieme, že

aktivita j má predchodcu aktivitu i, takže si túto informáciu uložíme do zoznamu predchodcov *„Predecessors“* aktivity j. Zatiaľ však máme pridané len cesty do priamo nasledujúcich aktivít, preto potrebujeme dopočítať všetky ostatné existujúce cesty. Tie dokážeme určiť pomocou jednoducho modifikovaného Floyd Washall algoritmu [1], ktorý slúži na nájdenie najmenšej cesty v orientovanom grafe s hranami kladných váh. Tento algoritmus bude pracovať s troma cyklami o veľkosti n, čo je opäť počet všetkých aktivít projektu. Indexy cyklov si označíme nasledovne, *k, i, j*, v tomto poradí. Pokiaľ máme v našej sieti s doposiaľ spracovanými cestami d, pre kombináciu i a k alebo k a j inú hodnotu, ako mínus nekonečno, a zároveň súčet týchto dvoch hodnôt je väčší ako je doposiaľ uložená hodnota pre dvojicu i a j, nahradíme ju týmto súčtom, $d_{ij} := \max(d_{ik} + d_{kj}, d_{ij})$. Týmto spôsobom dostaneme sieť s najdlhšími vzdialenosťami medzi aktivitami. Prvý riadok z d_0 nám dá rozvrh *ES*, ktorý spĺňa časové oneskorenia. Pomocou *UB* a d dokážeme určiť rozvrh najneskorších začiatkov aktivít *LS*, kde začiatok aktivity i je daný ako rozdiel hornej hranice a hodnoty posledného stĺpca z riadka aktivity, $LS_i := UB - d_{in}$. Dopočítanie hodnôt siete d popísaným spôsobom a nastavenie všetkých spomínaných hodnôt máme implementované v metóde *Prepare*.

Nakoniec si ešte pripravíme metódu *IsTimeFeasible*, ktorá prijíma rozvrh S, ktorým otestujeme, či spĺňa všetky časové oneskorenia. Pripravíme si pole *booleanov visited*, kde každý jeden záznam bude reprezentovať aktivitu a uchovávať informáciu o tom, či sme sa do tejto aktivity dokázali dostať. Začneme prechádzať všetky aktivity i a pre každú všetkých jej nasledovníkov j. Skontrolujeme, o koľko neskôr začína nasledujúca aktivita j od aktivity i. Ak je rozdiel ich začiatkov väčší, poprípade rovnaký, ako je nastavená cesta medzi nimi d_{ij}, $S_j - S_i \geq d_{ij}$ to znamená, že sa dokážeme dostať do nasledujúcej aktivity, $visited_j := true$. Na konci sa pozrieme, či boli navštívené všetky aktivity. Ak nie, vrátime si informáciu o tom, že vstupný rozvrh nespĺňa časové oneskorenia, inak vrátime pravdivostnú hodnotu, $S \in \mathcal{S}_T$.

4.2 Plánovanie projektu s obmedzenými zdrojmi

Keďže náš projekt využíva aj obnoviteľné zdroje, musíme si pripraviť metódy, ktoré nám budú slúžiť pre prácu s týmito zdrojmi. Samozrejme náš naplánovaný rozvrh bude musieť spĺňať aj obmedzenú kapacitu zdrojov. Aby sme to mohli overiť, budeme najskôr musieť určiť, ktoré aktivity sa v danú chvíľu vykonávajú. Na to nám bude slúžiť metóda *GetActiveInTime*, ktorá nám vráti zoznam aktívnych aktivít pre zadaný čas t a rozvrh S. Takýto zoznam sa nazýva aj aktívna množina rozvrhu S v čase t, $\mathcal{A}(S,t)$. Takýto zoznam budeme určovať nasledovne. Vytvoríme si pomocný zoznam triedy *ActivityStart*, ktorý bude na začiatku prázdny. Pre každú aktivitu i skontrolujeme, či je jej začiatok S_i menší alebo rovný času t, a zároveň je tento čas menší ako koniec aktivity. Ak je splnená táto podmienka, pridáme si do nášho zoznamu nový záznam, ktorý bude obsahovať informáciu o indexe aktivity i a jej začiatku S_i. Nakoniec si zoradíme všetky záznamy v našom zozname podľa začiatkov aktivít a vrátime si zoznam ich

identifikátorov.

$$\mathcal{A}(S,t) = \{i \in V \mid S_i \leq t < S_i + p_i\} \tag{2}$$

Keďže už vieme určiť aktívnu množinu, môžeme prejsť na testovanie splniteľnosti obmedzení na zdroje. Opäť budeme požadovať čas t a rozvrh S, pre ktorý chceme zistiť splniteľnosť. Najskôr si pre tieto vstupné parametre získame aktívne aktivity, $\mathcal{A}(S,t)$. Začneme prechádzať všetky naše obnoviteľné zdroje $\mathcal{R}$ a pre každý zdroj k si nastavíme celočíselnú premennú *sum* na nulu. Následne začneme prechádzať všetky aktivity $i \in \mathcal{A}(S,t)$. Pokiaľ aktivita používa zdroj,$r_{ik} > 0$ musíme spracovať túto informáciu. Ak sa čas t rovná času ukončenia aktivity, $t = S_i + p_i$ odpočítame toto množstvo r_{ik} z premennej *sum*, inak ho do nej pripočítame, $r_k(S,t) := \sum_{i \in \mathcal{A}(S,t)} r_{ik}$. Keď prejdeme všetky aktivity k zdroju k, skontrolujeme, či je celkový súčet používania zdroja väčší ako je jeho kapacita, $sum > R_k$. Ak áno, potom tento rozvrh nespĺňa obmedzenia zdrojov a vrátime si túto informáciu. Inak prechádzame na ďalší zdroj. Ak sme pre žiaden zdroj nezamietli tento rozvrh, potom je rozvrh zdrojovo splniteľný pre čas t. Takýto algoritmus máme implementovaný v metóde *IsResourceFeasible*, čiže pokiaľ budeme v texte spomínať overovanie zdrojov, bude sa myslieť tento spôsob.

$$r_k(S,t) \leq R_k \quad (k \in \mathcal{R}; t \geq 0) \tag{3}$$

Keby sme chceli zistiť, či je náš rozvrh splniteľný v každom čase, musíme ich všetky skontrolovať. Avšak pre efektívne zisťovanie overujeme len začiatky a konce aktivít, pretože vieme, že musia byť vykonané bez prerušenia. Na získanie všetkých týchto časov máme pripravenú metódu *GetTimes*, ktorá prijíma rozvrh, podľa ktorého chceme určiť časové body. Vytvoríme si zoznam celých čísel a začneme prechádzať všetky aktivity $i \in V$. Do nášho zoznamu si pridáme začiatok alebo koniec aktivity len v prípade, že ešte neobsahuje takúto hodnotu. Takýto zoznam časov bude výstupom z tejto metódy, pričom hodnoty sú zoradené vzostupne.

4.3 Fáza predspracovania

Všetky heuristické metódy, pomocou ktorých budeme riešiť problém plánovania projektov s obmedzenými zdrojmi, majú takzvanú fázu predspracovania. V tejto fáze sa zavedú do našej siete d s veľkosťami ciest medzi aktivitami ďalšie časové oneskorenia, vyplývajúce z konfliktov zdrojov dvojprvkovej zakázanej množiny F. Zakázanou množinou sa myslí taká množina dvojice aktivít, ktoré sa vykonávajú súčasne, a pritom presahujú celkovú kapacitu zdroja. To znamená, že pre daný rozvrh S, získame zakázanú množinu F, ako podmnožinu z aktívnej množiny, $F \subseteq \mathcal{A}(S,t)$. Ak je celkové potrebné množstvo niektorého zdroja k väčšie ako je jeho kapacita $\sum_{i \in F} r_{ik} > R_k (k \in \mathcal{R})$, potom je nutné priradiť medzi aktivity väčší rozostup. Nasledujúci teorém predspracovania hovorí o tom, ako budeme takýto konflikt zdrojov pre zakázanú množinu $F=\{i,j\}$ riešiť.

Teorém 2 *Ak pre dve aktivity i a j, $i \neq j$ a niektorý zdroj $k \in \mathcal{R}$ platí, že $r_{ik} + r_{jk} > R_k$ a (a) $LS_i < ES_j + p_j$ alebo (b) $d_{ij} >$ -p_j, potom musí platiť $S_j \geq S_i + p_i$ pre každý rozvrh $S \in \mathcal{S}$, pričom $S_n + 1 \leq UB$.*

Pridanie nového obmedzenia $S_j \geq S_i + p_i$ budeme vykonávať v metóde *Preprocessing*, pričom budeme prijímať hornú hranicu *UB*. Začneme prechádzať všetky aktivity *i* a pre každú všetky ostatné aktivity *j*, $i \neq j$ $(i, j \in V)$. Pre každú dvojicu aktivít *i,j* budeme prechádzať všetky zdroje $k \in \mathcal{R}$ a ako popisuje Teorém 2, skontrolujeme, či potrebné množstvo daného zdroja na vykonanie týchto aktivít presahuje jeho kapacitu, a zároveň platí varianta (a) alebo (b). Ak áno, musíme upraviť *d*. Pokiaľ doposiaľ neexistovala cesta medzi aktivitami $d_{ij} = -\infty$, dosadíme sem dobu trvania aktivity *i*, $d_{ij} := p_i$, inak vyberieme maximum z pôvodnej hodnoty a tejto doby trvania, $d_{ij} := \max(d_{ij}, p_i)$. Tento postup opakujeme, kým nastáva zmena v *d*. Potom si nastavíme nový rozvrh *ES*, *ES*=d_0 a ak je jeho celková doba trvania horšia ako je testovaná horná hranica *UB*, hovoríme, že *Preprocessing* zamietol *UB*. Pre takúto sieť musíme previesť test na splniteľnosť zadania, ktorý nám určí, či projekt obsahuje cykly pozitívnej dĺžky. To znamená, že pre každú dvojicu aktivít *i,j* skontrolujeme, či súčet vzdialeností medzi nimi je väčší ako nula, $d_{ij} + d_{ji} > 0$, ak áno, potom projekt nemá riešenie. Čiže návratová hodnota tejto metódy nám vracia informáciu o tom, či bola zamietnutá *UB* a či je možné pre plán nájsť splniteľný plán, alebo nie.

4.4 Dolná hranica trvania

Dolná hranica trvania projektu *LB* je veľmi dôležitou pri riešení RCPSP. Budeme určovať viacero druhov dolnej hranice. Napríklad prvú dolnú hranicu môžeme určiť z nášho rozvrhu *ES*, kde jej veľkosť bude daná dobou trvania tohto rozvrhu $LB0 := ES_{n+1}$, ktorá nám však zahŕňa len časové oneskorenia a nepočíta s dobou relaxácie pre zdroje. Ak by sme naopak chceli určiť dolnú hranicu na základe zdrojových obmedzení, dokážeme ju určiť tak, že pre každý zdroj *k* $(k \in \mathcal{R})$ a každú aktivitu *i* $(i \in V)$ spravíme súčet súčinov jej doby trvania p_i a požadovaného množstva daného zdroja touto aktivitou r_ik, ktorý podelíme celkovou kapacitou zdroja R_k a výsledkom bude najväčšia takto získaná hodnota pre zdroj $LBR := max_{k \in R} \sum_{i \in V} r_{ik} p_i / R_k$. Existuje aj takzvaná deštruktívna dolná hranica *LBD*, pre ktorú platí, že celková doba trvania splniteľného plánu je menšia ako táto hranica . Výpočet takejto hranice dokážeme pomocou techniky pólenia intervalu. Keďže všetky časové oneskorenia a doby trvania sú celé čísla, vždy existuje aj optimálny rozvrh, ktorý má tiež celočíselné hodnoty, a to za predpokladu, že daný projekt má riešenie. Algoritmus začína tak, že si nastavíme hodnotu začiatočnú dolnú hranicu *LB* výberom maxima z *LB0* a zaokrúhlenej hodnoty *LBR* nahor na celé číslo, *LB:*=max*(LB0,*$\lceil LBR \rceil$*)*, a ako hornú hranicu použijeme lokálnu hornú hranicu, *UB:*=$\overline{d}$ a pomocnú hodnotu *h*, ktorú nastavíme ako podiel súčtu *LB* a *UB* dvomi, ktorý zaokrúhlime nahor na celé číslo, $h := \lceil (LBR + UB)/2 \rceil$. Pre hodnotu *h* zavoláme metódu predspracovania *Preprocessing*, ak zamietne takúto hornú hranicu ako splniteľnú, skontrolujeme, či je rovnaká ako $\overline{d}$. Ak by sa rovnali, potom pre takýto

projekt neexistuje splniteľný plán. Ak sa nerovnajú, potom *LB := h+1* a znova vypočítame hodnotu *h*. V prípade, že nebola hodnota *h* zamietnutá ako vhodná horná hranica, zmenšíme ju o jedna a použijeme ju pre *UB*, *UB := h-1* a ak je doba trvania rozvrhu najskorších začiatkov ES_{n+1} väčšia ako hodnota *LB*, použijeme ho ako jej novú hodnotu, *LB:*=max(LB, ES_{n+1}) a vypočítame z nich novú hodnotu *h*. Tento proces opakujeme dovtedy, kým $LB \leq UB$. Takto získanú dolnú hranicu si uložíme do inštancie nášho projektu, do premennej LB. Nakoniec nám metóda vráti informáciu o tom, či bolo zamietnuté zadanie, alebo nie. Nakoniec si spomenieme dolnú hranicu, ktorá býva častokrát menšia ako doposiaľ spomínané hranice. Táto dolná hranica je úzko spätá s konceptom energetického uvažovania, čiže je určená využívaním zdrojov a budeme ju označovať ako *LBW*. Aby sme ju dokázali určiť, musíme si najskôr postup získania pracovnej záťaže pre zdroj *k* a rozvrh S^+ v čase *t'*, $w_k(S^+, t')$. Na to si pripravíme metódu *GetWorkload*. Tá nám pre všetky aktivity *i*, ktoré končia neskôr, ako je čas *t'*, vyberie menšiu hodnotu medzi dobou trvania aktivity p_i alebo jej začiatkom S_i^+ a z nich spraví celkový súčet.

$$w_k(S^+, t') = \sum_{\substack{i \in V \\ S_i^+ + p_i > t'}} r_{ik} \min(p_i, S_i^+ + p_i - t') \tag{4}$$

Keďže už vieme určiť pracovnú záťaž, môžeme pristúpiť k určeniu *LBW*. Vstupným parametrom bude rozvrh S^+ a na začiatku si nastavíme *LBW := 0*. Pre všetky zdroje k ($k \in \mathcal{R}$) a aktivity i ($i \in V$), si vypočítame w_k, pre zdroj k, rozvrh S^+ a čas *t'* bude začiatok danej aktivity, pričom vypočítanú pracovnú záťaž následne vydelíme celkovou kapacitou daného zdroja R_k a výsledok zaokrúhlime nahor na celé číslo. To následne pripočítame k začiatku aktivity. Ak je výsledok väčší ako je momentálna hodnota *LBW*, potom ju nahradíme týmto súčtom.

$$LBW(S^+) = \max_{k \in \mathcal{R}} \max_{i \in V} (S_i^+ + \lceil w_k(S^+, S_i^+)\ R_k \rceil) \tag{5}$$

Túto dolnú hranicu budeme využívať pri realizácií algoritmov typu *vetiev a hraníc*.

5 Postupy vetiev a hraníc

V tejto časti si popíšeme prvý druh algoritmov, riešiacich *RCPSP*, za pomoci postupu vetiev a hraníc (anglicky branch-and-bound). Táto technika sa všeobecne používa pre hľadanie optimálnych riešení rôznych optimalizačných problémov. Pracuje na princípe prechádzania všetkých možných kandidátov riešenia, pričom každý má svoje ohodnotenie. Môžeme si to predstaviť ako strom, kde každý vrchol je navrhovaný rozvrh riešenia a každý takýto vrchol ohodnotíme pomocou daného kritéria. Vždy začíname hľadať riešenie z vrcholu, ktorý má najlepšie ohodnotenie. Ak nájdeme splniteľný rozvrh projektu, odstránime naraz všetky vrcholy, ktoré majú horšiu hornú hranicu trvania, akú má toto riešenie. Takýto strom znázorňuje naša trieda *ActiveNodes* 3.2.6. Detailnú prácu so stromom si predstavíme v nasledujúcej podkapitole. V tejto kapitole si potrebne popíšeme pôvodný algoritmus *Enumerácia* a jeho skrátené varianty *Aproximačná metóda*, *Vyhľadávanie filtrovaním lúčov* a *VH Metóda rozkladu*.

5.1 Enumerácia

Ako prvú si popíšeme techniku enumerácie, ktorá je v mnohých prípadoch najpresnejšia, avšak časovo náročná, preto sa používa najmä na riešenie projektov s menším počtom aktivít.

Na začiatku si vytvoríme koreňový vrchol u typu *NodeItem* 3.2.5 nášho enumeračného stromu, ktorého rozvrhom bude rozvrh najlepších začiatkov $S^+ := ES$, o ktorom vieme, že je časovo splniteľný. Ako hornú hranicu zvolíme $\overline{d}$, ktorú zvýšime o jedna, *UB:=*$\overline{d}$ *+1*. Potom pre takúto hornú hranicu spustíme fázu predspracovania 4.3. Ak by ju táto zamietla *UB* ako vhodnú hornú hranicu, zistili by sme, že tento projekt nemá riešenie, $\mathcal{S} = \emptyset$, a algoritmus ukončíme. Inak vytvoríme novú inštanciu stromu, do ktorého vložíme pripravený vrchol u.
Následne môžeme zahájiť proces riešenia. Vyberieme prvý záznam zo zoznamu vrcholov nášho stromu u. Skontrolujeme, či rozvrh tohto vrcholu S^+ spĺňa všetky obmedzenia na zdroje, a to tak, že získame časy tohto rozvrhu *GetTimes* a pre každý čas overíme, či nie je prekročená kapacita zdrojov *IsResourceFeasible*. Ak zistíme, že rozvrh spĺňa všetky obmedzenia zdrojov, potom sme našli splniteľný plán projektu a uchováme si túto informáciu *Feasible := true*. Skontrolujeme, či doposiaľ zistená horná hranica projektu je väčšia ako je horná hranica spracovaného vrcholu. Ak áno, uchováme si ju ako novú najlepšiu nájdenú hornú hranicu projektu *BestUb*, a taktiež jeho rozvrh do premennej *BestScheduleSoFar*. Taktiež môžeme odstrániť všetky vrcholy stromu, ktorých horná hranica trvania je väčšia ako práve nájdená, metóda *Cut*. Pokiaľ by sme však zistili, že rozvrh nevyhovuje obmedzeniam zdroja, použijeme najskorší čas t, ktorý nevyhovel $r_k(S^+, t) > R_k$. Potom pre tento čas a rozvrh určíme aktívnu množinu $\mathcal{A}(S^+, t)$. Pre zoznam aktívnych aktivít prevedieme permutáciu bez opakovania, aby sme mali pokryté všetky možné varianty poradia aktivít *permutations*, ktoré budeme mať uchované ako zoznam zoznamov identifikátorov aktivít. Tieto možnosti budú naším vstupným parametrom metódy *GetResourceConflict*, ktorá nám vráti všetky konflikty medzi aktivitami v podobe zoznamu

pravidiel zdrojov *ResourceRules*. Jej algoritmus vyzerá nasledovne. Začneme prechádzať všetky prvky zoznamu možností *perm* a pre každý jeden vytvoríme novú inštanciu pravidla *rule* a pomocné pole o veľkosti počtu zdrojov *sum*, do ktorého budeme pripočítavať pre daný zdroj jeho využitie. Následne začneme prechádzať všetky aktivity $i \in perm$ v danom poradí a vytvoríme si pomocnú premennú *canAdd* typu *boolean*, nastavenú na pravdivú hodnotu, ktorá nám bude určovať, či je možné pridať aktivitu do zoznamu A pravidla *rule*. Pre každú aktivitu i budeme prechádzať všetky zdroje k a pripočítame do premennej *sum* množstvo zdroja potrebného na jej vykonanie $sum_k+ = r_{ik}$. Následne skontrolujeme toto množstvo. Ak by sme zistili, že je prekročená celková kapacita zdroja $R_k < sum_k$, zmeníme pravdivostnú hodnotu premennej *canAdd* a ukončíme prechádzanie zdrojov pre túto aktivitu. Po ukončení cyklu zdrojov skontrolujeme hodnotu *canAdd*. Ak je pravdivá, môžeme priradiť aktivitu i do zoznamu A, inak ju pridáme do zoznamu B pravidla *rule*. Toto pravidlo hovorí o tom, že aktivita j z množiny B môže začať až po najskoršom skončení aktivity i z množiny A.

$$\min_{j \in B} S_j \geq \min_{i \in A}(S_i + p_i) \tag{6}$$

Toto pravidlo nazývame aj disjunktívna prednosť, ktorá je označovaná ako $A \rightarrow B$.

Takýto postup však môže byť časovo veľmi náročný pre veľké projekty, ktoré majú viac ako 100 reálnych aktivít, nakoľko môže nastať situácia, kedy vznikne konflikt medzi veľkým počtom aktivít. Preto sme sa rozhodli použiť optimalizáciu určovania týchto pravidiel nasledovne. Pokiaľ zakázaná množina obsahuje aspoň desať aktivít, vytvoríme len toľko pravidiel, koľko je v nej aktivít. Začneme prechádzať všetky jej aktivity i a pre každú vytvoríme nové pravidlo *rule* tak, že opäť prejdeme všetky jej aktivity j. Pokiaľ sa $i = j$ pridáme aktivitu j do množiny B, inak ju pridáme do množiny A tohto pravidla *rule*. Nakoniec si vrátime zoznam takto vytvorených pravidiel. Samozrejme, pokiaľ je $|\mathcal{A}(S^+, t)| < 10$, postupujeme pôvodným spôsobom.

Pre každé získané pravidlo vytvoríme nový vrchol v, ktorého rozvrh S^v však musí vyriešiť zistený konflikt medzi aktivitami, aj všetky predchádzajúce pravidlá, a zároveň musí dodržať časové oneskorenia $\mathcal{S}(v) := \mathcal{S}(u) \bigcap \{S \in \mathcal{S}_T | \min_{j \in B} S_j \geq \min_{i \in A}(S_i + p_i)\}$. Na takéto získanie rozvrhu použijeme metódu *SetResourceLags*. Vstupom je práve spracovávaný vrchol u a pravidlo *rule*, pre ktoré chceme vyriešiť konflikt zdrojov. Pre každý vrchol si vždy uchovávame všetky predchádzajúce pravidlá v jeho vlastnosti *ParentConflicts*, ktoré sme pre jeho vetvu zistili. Pripravíme si nový rozvrh *S(v)*, ktorého hodnoty nastavíme podľa rozvrhu pôvodného vrcholu *S(u)*. Začneme prechádzať všetky doposiaľ zistené pravidlá pre u, vrátane nového pravidla *rule*, a pre každé vyberieme minimálnu dobu ukončenia aktivity $i \in A$ v práve určovanom rozvrhu S^v. Ak by bola táto minimálna hodnota horšia ako je doposiaľ najlepšia nájdená horná hranica, vrátime prázdny zoznam. Inak z množiny B vyberieme aktivitu s najskorším začiatkom a skontrolujeme, či je jej začiatok väčší ako predchádzajúce nájdené minimum $\min_{i \in A}(S_i + p_i)$. Ak áno, tak toto

minimum použijeme ako nový začiatok. Tento postup opakujeme, pokiaľ sa mení rozvrh S^v. Následne musíme pre takýto rozvrh zaviesť časové oneskorenia aktivít. Pre všetky aktivity h nášho projektu si vyberieme maximálnu hodnotu zo súčtu začiatku tejto aktivity S_h^v a dĺžky medzi aktivitami z množiny B a aktivitou h. Ak je táto maximálna hodnota väčšia ako je doposiaľ najlepšia nájdená horná hranica, vrátime si prázdny rozvrh. Inak skontrolujeme, či je táto hodnota väčšia ako začiatok aktivity h. Pokiaľ áno, použijeme ju ako jej nový začiatok $S_h^v := \max[S_h^v, \max_{j \in B}(S_j^v + d_{jh})](h \in V)$. Tento proces opakujeme, kým nastávajú v rozvrhu zmeny. Pokiaľ sme pri vytváraní nového rozvrhu nevrátili prázdny zoznam, metóda nám vráti rozvrh S^v.

Pokiaľ získaný nový rozvrh S^v pre spracovávané pravidlo *rule* nie je prázdny, vypočítame dolnú hranicu podľa vyťaženia zdrojov *LB(u):=LBW(*S^+*)* (5). Ak je táto dolná hranica menšia ako najlepšia nájdená horná hranica, a zároveň je menšia aj doba trvania rozvrhu S^v, môžeme pridať vrchol v do stromu. Chýba nám však ešte dôležitá informácia pre vrchol, a to je jeho ohodnotenie, podľa ktorého vyberáme najpravdepodobnejší vrchol blížiaci sa k riešeniu. Postup jeho určenia si teraz podrobne popíšeme. Budeme pracovať s hodnotou λ, ktorú si bude môcť používateľ zvoliť pred začiatkom algoritmu. Hodnota lambda môže byť ľubovolné číslo, väčšie ako nula a menšie ako jedna. Táto hodnota určuje, či chceme klásť väčší dôraz na čas alebo na zdroje. Ak sa hodnota blíži viac k 1, je pre nás dôležitejšie časové oneskorenia projektu. Ak sa blíži viac k 0, kladie väčší dôraz na zdroje. Ďalej budeme používať dva rozvrhy, medzi ktorými predpokladáme zlepšenie. Pre nás to bude rozvrh S^v a najlepší nájdený rozvrh, ktorý si označíme ako S^+. Začneme prechádzať všetkými zdrojmi k projektu a pre každý všetkými aktivity i, pričom si spravíme pomocú premennú uchovávajúci súčet súčinov medzi trvaním aktivity p_i, potrebným množstvom zdroja ne jej realizáciu r_{ik} a rozdielom jej začiatkom v novom predpokladanom zlepšenom rozvrhu S_i^v a začiatkom v rozvrhu S_i^+. Takto vypočítaný medzisúčet, vydelíme kapacitou zdroja R_k a spravíme si súčet týchto podielov pre všetky zdroje projektu, čím dostaneme celkový súčet. Výsledné ohodnotenie vrcholu získame tak, že odpočítame od 1 hodnotu λ, potom týmto rozdielom vynásobíme nami získaný celkový súčet a k výsledku pripočítame násobok doby trvania a λ.

$$\lambda S_{n+1}^v + (1-\lambda) \sum_{k \in \mathcal{R}} \frac{1}{R_k} \sum_{i \in V} (S_i^v - S_i^+) p_i r_{ik} \quad (0 < \lambda < 1) \tag{7}$$

Výpočet tohto ohodnotenia obsahuje metóda *GetNodeWeigh*. Keďže už dokážeme určiť všetky potrebné parametre nového vrcholu v, môžeme si ho pridať do stromu s rozvrhom S^v, pričom hornou hranicou bude doba trvania tohto rozvrhu S_{n+1}^v a ohodnotením určeným pomocou metódy (7). Po každom spracovaní vrcholu u je nutné ho zo stromu odstrániť, čo vykonáme metódou *Remove*, a následne musíme záznamy stromu zotriediť podľa ohodnotenia metódou *Order*, aby sme mali na prvej pozícii vrchol s najväčšou pravdepodobnosťou nájdenia riešenia. Tento postup spracovávania prebieha dovtedy, kým máme v strome nespracované záznamy. Algoritmus 1 sumarizuje enumeračnú metódu.

Algoritmus 1 Metóda vetiev a hraníc

Krok 1. Inicializácia koreňového vrcholu u s rozvrhom S^+ podľa *ES*. Nastavenie prehľadávajúceho priestoru $\mathcal{S}(u) := \mathcal{S}_\mathcal{T}$ a hornú hranicu $UB := \overline{d} + 1$. Vykonanie fázy predspracovania. Ak predspracovanie zamietne UB ako vhodnú hornú hranicu, potom skončíme (neexistuje splniteľný rozvrh).

Krok 2. Ak je S^+ zdrojovo splniteľný, nastavíme $UB := S^+_{n+1}$, odstránime horšie vrcholy a uchovávame si toto riešenie. Inak nájdeme najskorší čas t, v ktorom nastáva konflikt.

Krok 3. Pre každý oddiel $\{A,B\}$ z $\mathcal{A}(S^+, t)$, kde A je maximálna splniteľná množina a B je minimálna oneskorená alternatíva, definujeme nový vrchol v, kde $S(v) := S(u) \bigcap \{S \in \mathcal{S}_T | \min_{j \in B} S_j \geq \min_{i \in A}(S_i + p_i)\}$. Ak je výsledkom prázdna množina, alebo $S^v_{n+1} \geq UB$, zahodíme vrchol v.

Krok 4. Ak všetky nové vrcholy v boli odstránené, ukončí sa algoritmus. Inak nastavíme nový vrchol u, ktorý má minimálne ohodnotenie na základe (7) a $S^+ := S^u$. Určíme hodnotu dolnej hranice $LB(u) = LBW(S^+)$. Ak je $LB(u) \geq UB$,zahodíme vrchol u a ukončíme algoritmus. Inak ideme na Krok 2.

Po ukončení tohto algoritmu skontrolujeme, či náš najlepší nájdený rozvrh spĺňa všetky časové oneskorenia *IsTimeFeasible* a zdrojové obmedzenia *IsResourceFeasible*. Ak áno, výslednú dobu trvania projektu dostaneme na základe väčšej dolnej hranice $UB := \max(LBW, LB)$. Nasledujúce algoritmy tejto kapitoly, budú skrátenou verziou enumeračného algoritmu. Čo znamená, že budu používať rovnaký postup avšak s rôznymi úpravami, aby algoritmus dokázal nájsť riešenie za kratšiu dobu, čo však môže mať za následok to, že daný rozvrh nebude najlepší možný pre daný projekt.

5.2 Aproximačná metóda

Algoritmus vetiev a hraníc popísaný v podsekcii 5.1 môžeme jednoducho skrátiť na $\varepsilon-aproximačnú$ heuristiku, kde $\varepsilon > 0$. Túto hodnotu si bude môcť zvoliť používateľ.

Na začiatku zvolíme hornú hranicu UB ako $(1 + \varepsilon)\overline{d} + 1$. V kroku 4 budú vrcholy u rozvetvené iba ak spĺňajú takúto nerovnosť

$$(1 + \varepsilon)LB(u) < UB \tag{8}$$

Keďže máme hornú hranicu stanovenú spôsobom a $LB(u) \leq \overline{d}$, potom pre všetky vrcholy u, ktoré nemajú prázdny prehľadávajúci priestor, neexistuje vrchol u s $\mathcal{S}(u) \neq \emptyset$, až pokiaľ nenájdeme prvý splniteľný rozvrh S^+. Potom tak, ako predtým, nastavíme $UB := S^+_{n+1}$.

Predstavme si, že f^* označuje skutočnú hodnotu najlepšieho nájdeného plánu s použitím podmienky (8) pre vyhotovenie nových vrcholov z u a f^+ označuje skutočnú minimálnu hodnotu. Predpokladáme, že prehľadávajúci priestor $\mathcal{S}(u)$ pre aktivitu u obsahuje optimálny rozvrh. Potom platí $LB(u) \leq f^*$. Ak je vrchol u splniteľný, dostaneme $f^+ \leq UB \leq (1+\varepsilon)LB(u) \leq (1+\varepsilon)f^*$,

kde *UB* označuje aktuálnu hornú hranicu pri odstraňovaní vrcholu u z vyhľadávajúceho stromu. V takomto prípade relatívna odchýlka $(f^+ - f^*)/f^*$ nebude väčšia ako ε. Ak vetvíme z takéhoto vrcholu u, v ktorom existuje riešiteľný potomok, vrchol v, potom prehľadávaný priestor $\mathcal{S}(v)$ obsahuje optimálne riešenie alebo ho bude generovať, t.j. $(f^+ - f^*)/f^* = 0 < \varepsilon$. Takto popísaná skrátená verzia algoritmu vetiev a hraníc predstavuje ε-aproximačnú metódu. Je zrejmé, že časová zložitosť tohto postupu nie je polynomiálna.

5.3 Vyhľadávanie filtrovaním lúčov

Výpočet plánu enumeračnou alebo ε-aproximačnou metódou môže byť pre veľké projekty, ktoré majú stovky aktivít príliš časovo náročný. Preto si v nasledujúcej časti popíšeme ďalší zo skrátených algoritmov vetiev a hraníc, nazývaný *vyhľadávanie filtrovaním lúčov (anglicky filtered beam search)*. Tento algoritmus bude pracovať s takzvanou šírkou filtra φ a šírkou lúča β, ktorá musí byť menšia ako šírka filtra $\beta < \varphi$. Obe šírky sú celé nezáporné číslo. Tento algoritmus je založený na tom, že nespracúvame všetky možné získané vrcholy so všetkými ich zistenými pravidlami. To znamená, že po vygenerovaní všetkých podriadených vrcholov v z aktuálneho vrcholu u v kroku 3 algoritmu 1, spracujeme len pravidlo získané z tohto vrcholu, namiesto všetkých doposiaľ určených pravidiel pre danú vetvu. Preto si musíme rozšíriť metódu *SetResourceLags* o voliteľný parameter *currentOnly*, ktorého pravdivostná hodnota nám určí, či chceme spracovávať len aktuálne nájdené pravidlo alebo všetky pravidlá. Následne si vytvoríme vrchol v, ktorý však musí dodržať disjunktívnu prednosť $A \rightarrow B$. Všetky takéto vrcholy v pre spracovávaný vrchol u si odložíme do pomocného zoznamu, a ten nakoniec zoradíme od najkratšej doby trvania rozvrhov, pričom túto informáciu máme uloženú ako *UB* pre každý vrchol. Pre prvých φ takto zoradených vrcholov v z u spracujeme všetky zistené pravidlá pre danú vetvu a ostatné vrcholy nebudeme brať do úvahy. Nakoniec len počet β vrcholov s najkratšou dobou trvania priradíme do enumeračného stromu. Pre ťažké problémy môže byť šírka lúča s hodnotou 2 príliš veľká. Preto ju nebude možné zadať manuálne, ale vypočítame si ju. Nech $\widetilde{\beta} := \lceil \overline{\beta}\widetilde{u} \rceil$ označuje náhodné celé číslo, kde $\widetilde{u}$ je rovnomerne volené z intervalu $]0, 1]$ a $\overline{\beta} \geq 1$ je maximálna zvolená šírka lúča. Pre šírku lúča β potom zvolíme hodnotu $\widetilde{\beta}$. Ak je však hodnota $\overline{\beta} \leq 2$, potom vypočítame $\widetilde{\beta}$ ako 2-1/$\overline{\beta}$. Šírku filtra φ a maximálnu šírku lúča β budeme nastavovať podľa počtu aktivít projektu n a nastavením hodnoty σ, ktorá označuje maximálne požadované množstvom listov v našom strome. V našej implementácii je zvolená šírka filtra φ ako $\lceil \ln n \rceil$. Výpočtové experimenty s algoritmom vetiev a hraníc ukázali, že hĺbka stromu môže byť všeobecne ohraničená ako $n \ln n$. Preto sme zvolili $\overline{\beta} := 1/(2 - \sigma^{1/(n \ln n)})$. Na získanie tejto hodnoty β nám slúži metóda *GetBeam* v triede *FilteredBeamSearch*, pričom ju volíme pre každý spracovávaný vrchol u zvlášť. Z toho dôvodu, môže nastať situácia, že daný algoritmus vyhotoví pre jeden projekt rôzne rozvrhy.

5.4 Metóda rozkladu

V tejto podkapitole si popíšeme heuristiky rozkladu, ktoré sú prispôsobené efektívnemu výpočtu riešení na základe Teorému 1. Pred samotným celkovým riešením si najskôr zistíme všetky cykly C v našom projekte pomocou metódy *DetectCycle* z knižnice tried obsahujúcej Tarjanov algoritmus určovania silných komponentov. Ak je komponent tvorený minimálne 2 vrcholmi, dostávame cyklus C, pričom takýto cyklus bude pre nás označovať takzvaný podproblém.To znamená, že bude obsahovať len aktivity cyklu C, podľa ich definície, a samozrejme musíme obsahovať aj začiatočnú a koncovú aktivitu V^C. Tento podprojekt bude mať k dispozícií rovnaké zdroje, aké máme definované pre hlavný projekt. Pre každý takto definovaný podprojekt spustíme aproximačnú metódu. Ak by sme pre niektoré takéto zadanie projektu nenašli splniteľný rozvrh, potom nemá riešenie ani originálny projekt. Inak na základe podriešenia S^C zoradíme všetky aktivity $i \in V^C$ vyriešeného cyklu C od najskoršieho začiatku S_i^C. Pre každú dvojicu takto zoradených aktivít $i,j \in V^C$, priradíme nové minimálne časové oneskorenie, ak rozdiel ich začiatkov je väčší alebo rovný nule $d_{ij}^{min} := S_j^C - S_i^C \geq 0$. Okrem toho pridáme aj maximálne časové obmedzenie $d_{ij}^{max} := S_j^C - S_i^C \geq 0$ medzi začiatkom prvej aktivity $i \in V^C$ a poslednou aktivitou $j \in V^C$, čo zaručí, že všetky aktivity cyklu C sú pevne zviazané. Po spracovaní všetkých cyklov môžeme začať riešiť celkové zadanie, a to za pomoci mierne modifikovaného algoritmu vyhľadávanie filtrovaním lúčov. Pre takto upravený projekt budeme používať všetky disjunktívne prednostné obmedzenia $A \rightarrow B$, ktoré boli zavedené na ceste od koreňa až po aktuálny vrchol, oproti používaniu len práve spracovávaného pravidla. Vzhľadom k pridaniu minimálnych časových oneskorení, aktivity zakázanej množiny F, ktoré patria do jedného cyklu C, sa nebudú viac vykonávať súčasne, pretože tento problém sme už vyriešili. Po zistení splniteľnosti plánu pokračujeme pomocou pôvodného postupu. Z tohto dôvodu sme rozšírili algoritmus vyhľadávania filtrovaním lúčov tak, že môžeme nastaviť, či chceme použiť iba disjunktívne prednostné pravidlo zistené pre aktuálny vrchol alebo všetky, prostredníctvom vlastnosti triedy *FirstOnly*, pričom po nájdení riešenia zmeníme nastavenie tak, aby algoritmus fungoval klasickým spôsobom. Takto popísaný algoritmus budeme označovať ako *VH metóda rozkladu 1.* Jednoduchou úpravou dostaneme *VH metódu rozkladu 2.* Stačí, ak odstránime takzvané „spiatočné oblúky“, čo je vlastne prepojenie poslednej aktivity j s prvou aktivitou i pre každú štruktúru cyklu C. Čiže nastavíme hodnotu medzi týmito aktivitami na $d_{ji} := -\infty$. Aby sme nemuseli robiť dve implementácie tohto algoritmu, použijeme vlastnosť triedy *RemoveBackward*, ktorá nám bude hovoriť o tom, ktorý spôsob sa má realizovať.

6 Postupy používajúce prioritné pravidlá

V tejto kapitole si popíšeme algoritmy, ktoré vytvárajú plán projektovým aktivitám na základe prioritných pravidiel, avšak nie vždy poskytnú riešenie, aj keď nejaké existuje $\mathcal{S} \neq \emptyset$. Budeme používať dva rozdielne typy algoritmov využívajúcich techniky prioritných pravidiel. Takzvanú *priamu metódu*, ktorá bude spracovávať aktivity projektu jednu za druhou a *PP metódy rozkladu* 6.3, pričom *PP* je v tomto prípade skratka prioritného pravidla. Oba tieto algoritmy budú pracovať s cyklami C projektu, pričom priama metóda ich nebude vyhodnocovať samostatne. Metódy rozkladu budú opäť pracovať na rovnakom princípe, aký využívajú VH metódy rozkladu, a znova budú využívať teorém rozkladu (Teorém 1), čiže najskôr budeme hľadať riešenie pre každý cyklus. Oba tieto spôsoby budú využívať pre riešenie algoritmus *schému generovania sériových plánov* 6.2. Aktivity budú plánované po sebe, pričom po naplánovaní aktivity i bude ďalšia aktivita j vybraná na základe jedného z prioritných pravidiel, ktoré si popíšeme v nasledujúcej podkapitole 6.1.

6.1 Prioritné pravidlá

Tieto pravidlá budeme využívať na úspešné plánovanie aktivít. V tomto procese budeme pracovať s troma podmnožinami aktivít V. Prvou bude množina aktivít, ktoré už sme naplánovali $\mathcal{C}$. Začneme tak, že sem vložíme začiatočnú aktivitu $\mathcal{C}$:={0}, keďže vieme, že začína v čase nula. Ďalej budeme potrebovať množinu nespracovaných aktivít $\overline{\mathcal{C}} := V \setminus \mathcal{C}$, a nakoniec množinu vhodných aktivít $\mathcal{E}$, ktorú budú tvoriť všetci spracovaní predchodcovia spracovaných aktivít $\mathcal{E} := \{j \in \overline{\mathcal{C}} | Pred^{\prec c}(j) \subseteq \mathcal{C}\}$. Aktivita j^*, ktorú budeme plánovať, sa vždy vyberá z množiny $\mathcal{E}$,$j^* \in \mathcal{E}$, podľa najväčšej priority $\pi(j^*)$, pričom väzby sú rozdelené na základe zvyšujúceho sa počtu aktivity, to je $j^* = \min\{j \in \mathcal{E} | \pi(j) = ext_{h \in \pi}(h)\}$, kde *ext* znamená minimálnu alebo maximálnu hodnotu, čo záleží na použitom pravidle. To znamená, že ak *ext* označuje minimum (alebo maximum), aktivita $j = j^*$, čo má najväčšiu prioritu, čiže má najmenšiu (alebo najväčšiu) hodnotu prioritného pravidla $\pi(j)$. Existuje veľké množstvo takýchto pravidiel [3], my sme si zvolili len päť z nich.

LST pravidlo (najmenší najneskorší začiatok - „latest start time“) vyberá aktivitu h ($h \in \mathcal{E}$) tak, že najskôr treba spracovať tú, ktorá má najskorší začiatok v rozvrhu najneskorších možných začiatkov LS. Prejdeme teda všetky nespracované aktivity h a vrátime si identifikátor tej, ktorá ma najmenšie číslo v rozvrhu LS, $ext_{h \in \mathcal{E}} \pi(h) = \min_{h \in \mathcal{E}} LS_h$.

MST pravidlo (minimálny voľný čas - „minumum slack time“) pracuje s takzvaným voľným časom aktivity h ($h \in \mathcal{E}$), ktorý získame tak, že od najneskoršieho možného začiatku aktivity odpočítame jej najskorší možný začiatok, $TF_h = LS_h - ES_h$. Opäť budeme pracovať len s nespracovanou množinou aktivít $\mathcal{E}$ a vyberieme tú aktivitu h, ktorá má najmenšiu takto vypočítanú hodnotu, $ext_{h \in \mathcal{E}} \pi(h) = \min_{h \in \mathcal{E}} TF_h$.

MTS pravidlo (najviac celkových nasledovníkov - „most total succesors“) vyberá aktivitu h ($h \in \mathcal{E}$) na základe maximálneho celkového počtu dosiahnuteľných nasledovníkov (nie len nevyhnutne nasledujúce aktivity). To znamená, do koľkých aktivít sa dokážeme z aktivity dostať, pričom je dodržané ich poradie $\prec$. Zoznam týchto nasledovníkov budeme nazývať dosiahnuteľnou množinou $Reach^{\prec}(h)$. Vrátime si aktivitu h, ktorá má najviac týchto nasledovníkov, $ext_{h\in\mathcal{E}}\pi(h) = \max_{h\in\mathcal{E}} |Reach^{\prec}(h)|$.

LPF pravidlo (najdlhšia nasledujúca cesta - „longest path following“) toto pravidlo pracuje s najväčším množstvom navštívených vrcholov $l(h)$, z cesty medzi aktivitou h ($h \in \mathcal{E}$) do koncovej aktivity projektu *n+1*. Pokiaľ sa rozhodneme použiť toto pravidlo, spočítame si pre každú aktivitu $i \in V$ najdlhšiu cestu do konca a uchováme si túto informáciu v zozname dĺžky cesty *NodesCountToFinish*, ktorý budeme používať pri vyhodnocovaní týmto pravidlom $ext_{h\in\mathcal{E}}\pi(h) = \max_{h\in\mathcal{E}} l(h)$.

RSM pravidlo (metóda plánovania zdrojov -„resource scheduling method“). Pre každú nespracovanú aktivitu h ($h \in \mathcal{E}$) a všetky ostatné aktivity g z tejto množiny, $g \in \mathcal{E} \setminus \{h\}$ si vyberieme maximálnu hodnotu z rozdielu najneskoršieho možného začiatku aktivity g a najskoršieho možného konca aktivity h, $\max(ES_h + p_h - LS_h)$. Ak je takáto maximálna hodnota záporná, nahradíme ju nulou a vyberieme prvú aktivitu, ktorá má najväčšie toto maximum, $ext_{h\in\mathcal{E}}\pi(h) = \min_{h\in\mathcal{E}} \max[0, \max_{g\in\mathcal{E}\setminus\{h\}}(ES_h + p_h - LS_g)]$.

Každé z týchto pravidiel je implementované ako metóda, ktorú si môže používateľ zvoliť a bude sa používať v algoritme, ktorý rieši problém pomocou schémy generovania plánu. Vstupným parametrom týchto metód je množina doposiaľ nespracovaných aktivít $\mathcal{E}$ a implementované sú v triede *PriorityRules*.

6.2 Schéma generovania plánu

V nasledujúcom texte si popíšeme schému generovania sériových plánov s použitím prioritného pravidla. Jej implementáciu budeme vykonávať v triede *PriorityRules*, budeme však potrebovať parametre, ktoré budeme využívať počas hľadania riešenia týmto spôsobom, a to lokálnu inštanciu najneskoršieho rozvrhu *LS* a lokálnu inštanciu najskoršieho rozvrhu *ES*, počet toho, koľkokrát sme vykonali krok „odplánovania“ u a dvojrozmerné pole na uchovanie profilov zdrojov. Pre vytvorenie novej inštancie je potrebné zadať projekt, všetky jeho nájdené cykly a pravidlo, ktoré budeme používať. Pre lokálne rozvrhy *LS* a *ES* použijeme ich hodnoty z projektu, *UB* nastavíme na lokálnu hornú hranicu trvania.

Samotný algoritmus riešenia prebieha nasledovne. Najskôr priradíme do množiny $\mathcal{C}$ začiatočnú aktivitu *0*. Ďalej si musíme určiť profily zdrojov. To znamená, že pre každý zdroj k ($k \in \mathcal{R}$) a každý časový bod τ musíme určiť množstvo, ktoré všetky aktivity z tohto zdroja spolu využívajú. Všetky časové body budeme získavať na základe nášho rozvrhu *LS*. Prejdeme množinu

zdrojov $\mathcal{R}$ a pre každý vytvoríme nové pole o veľkosti LS_{n+1}. Následne budeme prechádzať všetky už spracované aktivity $i \in \mathcal{C}$ a pre každú z nich budeme prechádzať všetkými časmi t jej trvania p_i. Keď ku každému času t pripočítame hodnotu z lokálneho rozvrhu ES_i pre aktivitu i, získame časový bod τ. Následne pre všetky zdroje k a časové body τ musíme pripočítať používané množstvo tohto zdroja aktivitou i, čím získame celkový súčet využívania zdroja v čase $r_k(\tau)$. Ide o metódu *SetResourceProfiles*. Keď už máme pripravené všetky profily zdrojov, môžeme pokračovať ďalej. Pripravíme si množiny $\overline{\mathcal{C}}$ spôsobom,ktorý sme si popísali v kapitole 6.1. Teraz môžeme zahájiť samotný výpočet, ktorý bude prebiehať, pokiaľ máme aspoň jednu nespracovanú aktivitu a nebol prekročený počet „odplánovacieho“ kroku. Určíme si množinu $\mathcal{E}$ a použijeme ju ako vstupný parameter pre nami zvolené prioritné pravidlo, ktoré nám vráti aktivitu j^* s najväčšou prioritou spracovania. Pre aktivitu j^* si nájdeme najskorší čas t^*, v ktorom nie je prekročená kapacita žiadneho zdroja. Určenie tohto času je nasledovné. Začneme prechádzať všetky časové body τ aktivity j^* od jej najskoršieho začiatku z nášho lokálneho rozvrhu $\tau := ES_{j^*}$ a budeme pokračovať až po jej najneskorší možný koniec podľa pôvodného rozvrhu $LS_{j^*} + p_j$. Pre každý τ skontrolujeme profil každého zdroja, či nebola prekročená jeho kapacita $r_k(\tau) \leq R_k$. Časový bod, ktorý neprekračuje kapacitu žiadneho zdroja $k \in \mathcal{R}$ bude našim výsledným časom t (metóda *GetMinTime*). Ak by bol takto určený čas horší, ako je hodnota lokálneho rozvrhu pre navrhovanú aktivitu $t^* > LS_{j^*}$, potom t^* nie je časovo splniteľný a je nutné previesť fázu odplánovania 6.2.1. Ak sme tento čas nezamietli, môžeme ho použiť ako vhodný začiatok aktivity $S_j^* := t^*$. Ďalej upravíme množinu $\mathcal{C}$, do ktorej pridáme aktivitu j^* a odoberieme túto aktivitu z množiny $\overline{\mathcal{C}}$. Potom musíme opäť aktualizovať profily zdrojov, štandardným postupom. Musíme však upraviť lokálne rozvrhy ES a LS pre všetky nespracované aktivity $i \in \overline{\mathcal{C}}$. Čiže pre každú aktivitu j nastavíme ES_j výberom väčšej hodnoty medzi jej pôvodnou hodnotu a súčtom jej začiatku s dĺžkou cesty z j^* do j, $ES_j := \max(ES_j, S_j + d_{j^*j})$ a hodnotu LS_j nastavíme minimom z pôvodnej hodnoty a rozdielom dĺžky cesty z j do j^*, $LS_j := min(LS_j, S_j - d_{jj^*})$. Tento postup však môžeme optimalizovať s použitím nasledujúceho pravidla. Po každom úspešnom naplánovaní aktivity j^* si vyberieme všetky aktivity jej cyklu V^C a upravíme výber aktivít do množiny $\mathcal{E}$. To znamená, že pokiaľ existujú aktivity cyklu, ktoré ešte neboli spracované, budeme vyberať z týchto aktivít. Inak postupujeme štandardným postupom výberu.

$$\mathcal{E} := \begin{cases} \{j \in \overline{\mathcal{C}} \bigcap V^C | Pred^{\prec c}(j) \subseteq \mathcal{C}\} & \text{ak niektora aktivita, avšak nie všetky, z cyklu } C \\ & \text{bola naplánovaná} \\ \{j \in \overline{\mathcal{C}} | Pred^{\prec c}(j) \subseteq \mathcal{C}\} & \text{inak} \end{cases} \tag{9}$$

6.2.1 Fáza odplánovania

Ako vstupné parametre pre túto metódu použijeme aktivitu j^* a čas Δ, ktorý získame ako $t^* - LS_{j^*}$. Všeobecne platí, že posledná doba začiatku LS_{j^*} vyplýva z maximálneho časového oneskorenia $d_{ij^*}^{max}$ medzi začiatkom niektorej aktivity $i \in \mathcal{C}$ a začiatkom aktivity j^*, tj. $LS_{j^*} := S_i - d_{j^*i}$. Preto si na začiatku určíme množinu $\mathcal{U}$, ktorá bude obsahovať aktivity, pre ktoré musíme zrušiť navrhovaný začiatok. Tú získame tak, že vyberieme z množiny $\mathcal{C}$ tie aktivity, pre ktoré platí $LS_{j^*} = S_i - d_{j^*i}$. Ak by takto určená množina obsahovala aj začiatočnú aktivitu *0* alebo je prekročený celkový počet vykonaní tohto kroku, pričom táto hranica je daná počtom aktivít projektu n, potom nemá daný problém riešenie a ukončíme algoritmus. Inak pre každú aktivitu $i \in \mathcal{U}$ nastavíme novú hodnotu najskoršieho začiatku $ES_i := S_i + \Delta$, odoberieme ju z množiny $\mathcal{C}$ a následne ju pridáme do $\overline{\mathcal{C}}$. Po vykonaní tohto kroku prejdeme všetky zostávajúce aktivity $i \in \mathcal{C}$ a ak je začiatok niektorej z nich S_i menší ako minimálny hodnota začiatku z aktivít množiny $\mathcal{U}$, $S_i < \min_{h \in \mathcal{U}} S_h$, potom ju musíme taktiež odobrať z $\mathcal{C}$ a pridať do $\overline{\mathcal{C}}$. Keďže sa nám zmenila množina spracovaných aktivít, je nutné znova previesť určenie profilov zdrojov.

6.3 Priama metóda a metóda rozkladu

Ako sme už spomínali, tento druh algoritmu bude pracovať so štruktúrou cyklov v projekte, ale nebude ich vyhodnocovať samostatne. Najskôr si musíme nájsť všetky štruktúry cyklu s využitím našej implementácie Tarjanovho algoritmu, rovnako ako sme to robili pre VH metódy rozkladu 5.4. Podobne, ako v predchádzajúcich algoritmoch, použijeme pre hornú hranicu trvania projektu vypočítanú lokálnu hornú hranicu $UB := \overline{d}$. Následne pre ňu prevedieme fázu predspracovania. Ak nebola táto horná hranica zamietnutá, môžeme pokračovať v hľadaní splniteľného rozvrhu, a to pomocou schémy generovania sériových plánov, pričom aktivity z rovnakého cyklu sú spracovávané postupne jedna za druhou. V tomto algoritme budeme využívať predchodcov aktivity, preto je nutné aplikovať metódu *SetPredecessors*.

Algoritmus 2 Priama metóda

Krok 1. Nájdeme všetky cykly C medzi aktivitami projektu.

Krok 2. Vypočítame $\overline{d}$ a použijeme ju ako hornú hranicu $UB{:=}\overline{d}$. Prevedieme fázu predspracovania s UB. Ak predspracovanie zamietne UB, ukončíme algoritmus (neexistuje splniteľný rozvrh).

Krok 3. Určíme predchodcov $Pred^{\prec}(i)$ pre všetky aktivity $i \in V$. Nájdeme splniteľný rozvrh pomocou schémy generovania sériového plánu, pre projekt, kde všetky aktivity jedného cyklu sú plánované jedna za druhou.

Teraz si popíšeme metódy rozkladu. Fungujú na rovnakom princípe pridávania dodatočných časových obmedzení, ako sme si popísali v sekcii hľadania riešenia pomocou metódy rozkladu, s použitím algoritmu vetiev a hraníc 5.4. Najskôr prevedieme rovnaký začiatočný postup, aký sme použili pri priamej metóde, čiže si určíme lokálnu hornú hranicu trvania $\overline{d}$ a prevedieme

základné testy na splniteľnosť projektu. Ak sme nezamietli tento projekt, môžeme pokračovať ďalej, a to tak, že pre každý jeden cyklus C si určíme predchodcov a skúsime preň nájsť riešenie postupom popísaným v kapitole 6.2. Rovnako, ako v predchádzajúcej metóde rozkladu, ak zistíme, že cyklus nemá splniteľný plán, potom celý projekt nemá riešenie, inak pridáme časové obmedzenie do našej projektovej siete. Nakoniec si pre takto upravenú sieť určíme predchodcov každej aktivity a spustíme proces plánovania schémou generovania sériového plánu. Taktiež budeme aj teraz mať dva druhy tejto metódy. To znamená, že *PP metódu rozkladu 2* získame tak, že odstránime spiatočné oblúky.

Algoritmus 3 PR metóda rozkladu 1

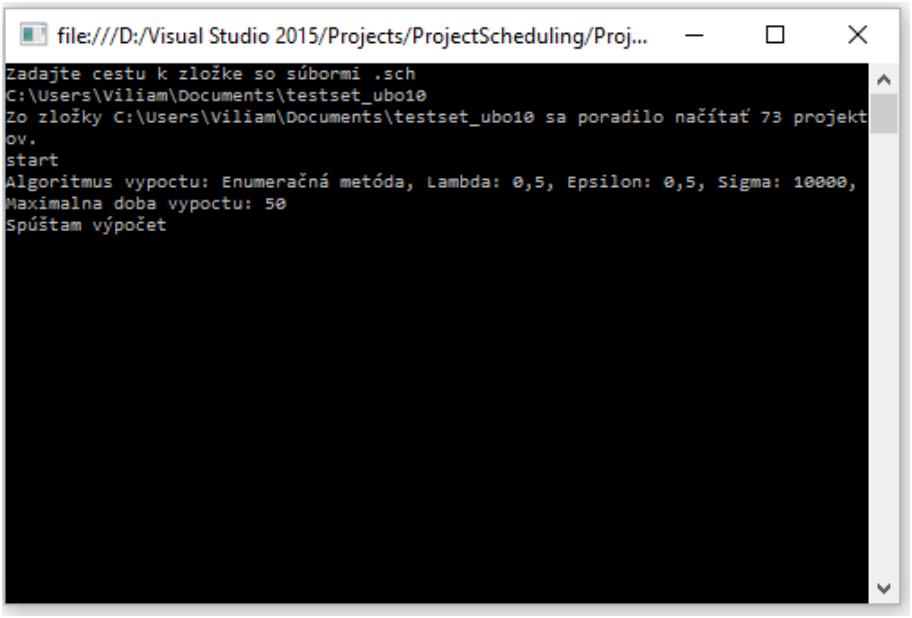
Krok 1. a krok 2. sú rovnaké, ako v priamej metóde (Algoritmus 2).
Krok 3. Vykonáme krok 2 a 3 priamej metódy pre každú štruktúru cyklu $\mathcal{C}$. Ak pre niektorý $\mathcal{C}$ nenájdeme splniteľný plán, potom neexistuje riešenie celého projektu a ukončíme algoritmus.
Krok 4. Pridáme časové obmedzenia na základe S^C do projektu.
Krok 5. Vykonáme krok 3 priamej metódy pre takto upravený projekt.

7 Spracovanie projektu

Naša aplikácia nevyžaduje veľkú interakciu s používateľom a môžeme ju chcieť opakovane spúšťať. Preto sme zvolili konzolovú aplikáciu programovacieho jazyka C#.

7.1 Načítanie a spracovanie zadania

Po spustení aplikácie sa nám v konzole vypíše správa „Zadajte cestu k zložke so súbormi .sch“. To znamená, že zadanie projektu sa bude načítavať zo súboru tohto typu. Po zadaní cesty k zložke, sa vyberú všetky súbory s príponou *.sch* a zotriedia sa podľa názvu. Ak by sa v tomto kroku niečo nepodarilo, alebo by zložka neobsahovala ani jedno zadanie, vypíše sa chybová správa „Nepodarilo sa načítať súbory z „vami definovaná cesta“. Pokiaľ však obsahovala tieto súbory,



Obr. 5: Aplikácia v stave po načítaní projektov a pri zahájení výpočtov

začneme ich všetky prechádzať a budeme sa ich snažiť spracovať, pričom si úspešne spracované zadania budeme odkladať do zoznamu projektov. Avšak súbory so zadaním projektu musia striktne dodržiavať stanovený formát, ktorý si teraz ukážeme.

Formát vstupného súboru

Pre spracovanie vstupného súboru do definovaných dátových štruktúr, sme si pripravili pomocnú triedu *FileReader*. V tejto triede máme metódu *ReadFromFile*, ktorej vstupným parametrom je celá cesta k súboru s definíciou projektu. Postupne načíta všetky riadky súboru pomocou systémovej knižnice „System.File“ metódou *ReadAllLines*, ktorá nám vráti zoznam textových reťazcov, pričom jeden riadok je jedným záznamom zoznamu. V riadku sú vždy informácie od-

n	ρ	0	0					
0	1	s_0	j_1^0	...	j_{s0}^0	$[\delta_{0,j_1^0}]$	...	$[\delta_{0,j_{s0}^0}]$
1	1	s_0	j_1^1	...	j_{s1}^1	$[\delta_{1,j_1^1}]$	...	$[\delta_{1,j_{s1}^1}]$
...								
n	1	s_n	j_1^n	...	j_{sn}^n	$[\delta_{n,j_1^n}]$	...	$[\delta_{n,j_{sn}^n}]$
$n+1$	1	0						
0	1	0	0	...	0			
1	1	p_1	$r_{1,1}$	...	$r_{1,\rho}$			
...								
n	1	p_n	$r_{n,1}$	...	$r_{n,\rho}$			
$n+1$	1	0	0	...	0			
R_1	...	R_ρ						

Symbol	Význam
n	Počet aktivít
ρ	Počet obnoviteľných zdrojov
s_i	Počet priamych nasledovníkov vrcholu i v projektovej sieti
j_j^i	s-tý nasledovník vrcholu i v projektovej sieti
δ_{i,j_s^i}	Váha oblúka (i, j_s^i)
p_i	Trvanie aktivity i
r_{ik}	Počet jednotiek zdroja k používaných aktivitou i
R_k	Kapacita zdroja k

delené pomocou tabulátora, čiže pri spracovaní, si riadok rozdelíme na textové pole pomocou metódy *Split*, podľa znaku *'t'*, ktorý znázorňuje tabulátor. Z formátu súboru vieme, že prvý riadok nám určuje, koľko má daný projekt reálnych aktivít n, čiže musíme pripočítať ešte začiatočnú a koncovú aktivitu, čím získame celkové množstvo aktivít a počet zdrojov ρ. Preto si potrebujeme túto informáciu prečítať zo súboru a uložíme si ju v novej inštancii *Project* do vlastnosti *„ActivityCount“*$= n$, *„RenewableCount“*$= \rho$. Na základe tejto informácie budeme vedieť, kedy treba riadok spracovávať ako aktivitu a kedy už sa jedná o definíciu módu aktivity. Začneme teda prechádzať všetky načítané riadky okrem prvého a posledného. Pokiaľ je index riadka menší, alebo rovný počtu aktivít n, vieme, že ho treba spracovávať ako aktivitu projektu, trieda *Activity*. Z každého takéhoto riadka si prečítame všetky informácie, ktoré definujú aktivitu, identifikátor, počet módov, ktorý musí byť v našom prípade vždy len jeden, počet bezprostredných nasledovníkov, podľa ktorého načítame identifikátory týchto nasledovníkov a váhy medzi nimi. Takto spracovanú aktivitu si pridáme do zoznamu aktivít *„Activities“* nášho projektu. Pokiaľ index riadka prekročí počet aktivít, vieme, že tieto riadky musíme spracovávať ako mód aktivity *Mode* a musíme ho priradiť k správnej aktivite. Mód musí mať definovaný identifikátor aktivity, ku ktorej patrí, počet módov, ktorý musí byť vždy 1, dobu trvania aktivity a ku každému zdroju definovaný počet potrebný na jej vykonanie, pričom ich poradie označuje identifikátor zdroja. Ak by aktivita niektorý zdroj nevyužívala, vtedy treba nastaviť na jeho pozíciu hodnotu 0, aby sa predišlo chybnému načítaniu zadania. Nakoniec načítame z posledného riadka celkovú kapa-

citu každého zdroja, pričom ich poradie opäť zodpovedá identifikátoru a ich počet musí súhlasiť s ρ . Túto kapacitu zdrojov si uložíme do zoznamu *„Renewables“* našej inštancie projektu. Keby sa pri spracovávaní v akomkoľvek kroku zistilo, že vstupný súbor nespĺňa požadovaný formát, metóda sa prestane vykonávať, vráti chybu, konzola vypíše, že daný súbor sa nepodarilo načítať a pokračuje v načítaní ďalšieho zadania, ak existuje. Pokiaľ sa nám nepodarilo načítať žiaden projekt, aplikácia upozorní používateľa správou, že sa nám z nej nepodarilo načítať žiaden projekt a vráti sa na krok zadania cesty k zložke. Inak nám vráti informáciu o tom, koľko zadaní sa z nami zadanej zložky podarilo načítať. Ak sa podarilo načítať aspoň jedno zadanie projektu, môžeme používať nasledujúce príkazy na ovládanie aplikácie, poprípade zadávanie parametrov týkajúcich sa hľadania riešenia.

Príkazy:

- h/p/help/pomoc/pomocnik - vypíše do konzoly stručný popis aplikácie, k čomu slúži, príkazy, pomocou ktorých sa dá ovládať, parametre, zoznam algoritmov a prioritných pravidiel, ktoré je možné použiť
- rr/restart/obnovit/obnov – vráti aplikáciu do pôvodného stavu, kedy je potrebné zadať cestu k zložke so zadaniami projektov
- start/spustit/spust – spustí algoritmus výpočtu splniteľného rozvrhu pre všetky načítané zadania
- set/nastav/nastavit – slúži k nastaveniu hodnoty parametra, pričom sa najskôr zadá názov parametra, a potom hodnota, pričom sú oddelené medzerou, napr. *set lambda 25*
- get/dostan/dostat – vráti nám informáciu pre požadovaný parameter, napr. *get lambda*

Parametre:

- output/path/vystup/cesta – kam sa má vygenerovať súbor s výsledkami projektov (prednastavená prázdna hodnota - nebude sa generovať výstupný súbor)
- duration/time/maxduration/maxd/trvanie/cas/maxtrvanie/maxt – ako maximálne dlho má vybraný algoritmus hľadať riešenie jedného problému v sekundách (prednastavený na 50 sekúnd)
- l/lambda – parameter λ, ktorý využívajú algoritmy typu Vetiev a hraníc, môže byť nastavený na hodnotu väčšiu ako nula a menšiu ako jedna (prednastavený na 0,5)
- e/epsilon – parameter ε, ktorý využívajú algoritmy používajúce ε-aproximačnú metódu, môže byť nastavený na ľubovoľnú číselnú hodnotu väčšiu ako nula (prednastavený na 0,75)
- s/sigma – parameter σ, používaný algoritmom vyhľadávanie filtrovaním lúčov, môže byť nastavený na ľubovoľné celé číslo väčšie ako 0 (prednastavený na 10000)

- pr/pp/rule/pravidlo - určí, aké prioritné pravidlo sa bude používať pre algoritmy využívajúce tieto pravidlá, hodnota je index zo zoznamu prioritných pravidiel (prednastavený na 0)
- alg/algo/algorithm/algoritmus – určuje, ktorý algoritmus sa bude používať pri riešení problémov, hodnota je index zo zoznamu algoritmov (prednastavený na 0)

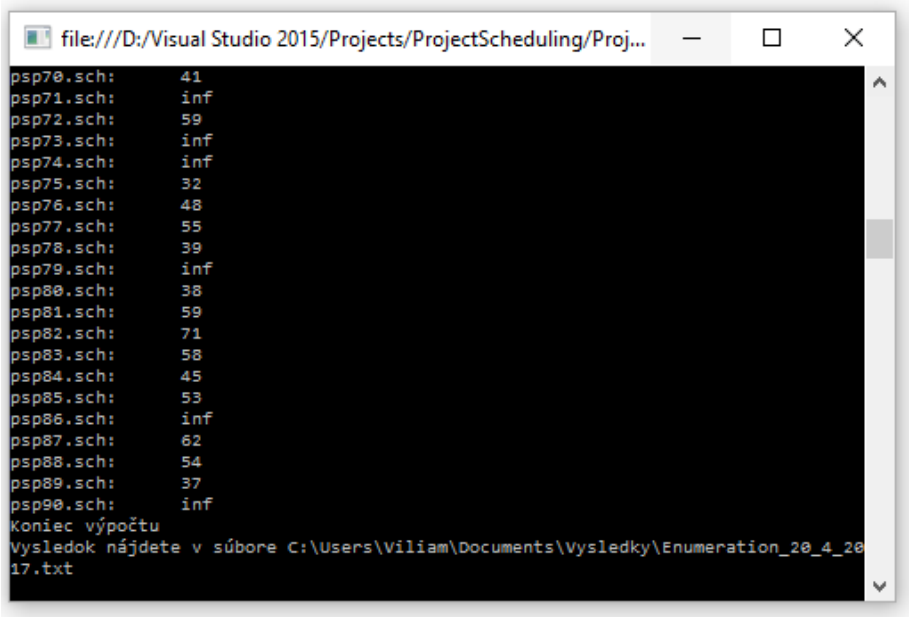
Zoznam algoritmov:

- 0 – Enumerácia
- 1 – Aproximačná metóda
- 2 – Vyhľadávanie filtrovaním lúčov
- 3 – VH metóda rozkladu 1
- 4 – VH metóda rozkladu 2
- 5 – Priama metóda
- 6 – PP metóda rozkladu 1
- 7 – PP metóda rozkladu 2

Zoznam prioritných pravidiel:

- 0 – LST - najmenší najneskorší začiatok
- 1 – MST - minimálny voľný čas
- 2 – MTS - najviac celkových nasledovníkov
- 3 – LPF - najdlhšia nasledujúca cesta
- 4 – RSM - metóda plánovania zdrojov

Výstupom z našej aplikácie bude informácia o hornej hranici projektu, pokiaľ preň bolo nájdené riešenie. Ak projekt nemá riešenie, alebo nebolo nájdené v stanovenom čase, bude vrátený výsledok *inf*, označujúci nekonečno. Takáto informácia sa uloží aj do výstupného súboru, pričom tento súbor bude naviac obsahovať informáciu o tom, aký bol použitý algoritmus a ako boli nastavené parametre aplikácie. Výstupný súbor je vo formáte textového súboru a jeho názov je zvolený podľa použitého algoritmu a dátum vykonania riešenia (algoritmus_den_mesiac_rok.txt).



Obr. 6: Aplikácia v stave po ukončení výpočtov

8 Testovanie algoritmov

Nakoniec prevedieme niekoľko testov algoritmov, aby sme porovnali ich efektivitu. Testovať budeme projekty používajúce $n \in \{10, 20, 50, 100\}$ reálnych aktivít, pričom pre každú máme pripravených 90 rôznych zadaní. Každý projekt bude využívať 5 obnoviteľných zdrojov. Z takejto množiny projektov si najskôr musíme určiť, koľko z nich pre každú množinu je reálne riešiteľných. To zistíme pomocou fázy predspracovania, ktorá odhalí splniteľnosť zadania, či niektorá aktivita nepožaduje väčšie množstvo zdrojov ako je jeho celková kapacita, alebo či projekt neobsahuje cykly pozitívnej dĺžky. Výsledky môžeme vidieť v tabuľke 1.

Tabuľka 1: Percento riešiteľných projektov testovaných množín

$n = 10$	$n = 20$	$n = 50$	$n = 100$	$n = 200$
81%	77%	81%	87%	89%

Zadania, ktoré nemajú riešenie vyradíme z testovacej množiny a budeme testovať už len zadania, ktoré sú splniteľné. Pre takúto množinu vykonáme niekoľko testov. V prvom teste si porovnáme schopnosť algoritmov nájsť riešenie v maximálnom čase 50 sekúnd, pričom čas sa začne merať až od zahájenia samotného algoritmu. Každý projekt sme testovali s rôznymi nastaveniami algoritmov, aby boli naše výsledky čo najpresnejšie. V nasledujúcich testoch však nebudeme používať enumeračný algoritmus vetiev a hraníc, ale budeme porovnávať všetky jeho skrátené verzie, aby sme zistili, či dokážu byť tak efektívne, ako tento algoritmus. Pre skrátené algoritmy typu vetiev a hraníc sme používali všetky možné kombinácie z nastavení hodnôt $\lambda \in \{0.25, 0.50, 0.75\}$ a $\varepsilon \in \{0.5, 1, 1.5\}$. Pre algoritmy používajúce vyhľadávanie filtrovaním lúčov sme nastavili maximálny počet vetiev $\sigma = 10000$. Pre algoritmy využívajúce prioritné pravidlá, sme použili kombináciu všetkých implementovaných pravidiel pre každý algoritmus. Výsledky testu máme znázornené v tabuľke 2.

Tabuľka 2: Úspešnosť algoritmov v zhotovení splniteľného plánu

	$n = 10$	$n = 20$	$n = 50$	$n = 100$
Aproximačná metóda	100%	95%	63%	49%
Vyhľadávanie filtrovaním lúčov	98%	97%	65%	36%
VH metóda rozkladu 1	100%	95%	68%	55%
VH metóda rozkladu 2	100%	96%	76%	64%
Priama metóda	98%	100%	98,63%	85,99%
PP metóda rozkladu 1	100%	100%	97,26%	83,33%
PP metóda rozkladu 2	100%	100%	97,26%	85,99%

Podľa výsledkov je zrejmé, že s narastajúcim počtom aktivít a obmedzením na maximálnu dobu trvania výpočtu 50 sekúnd, sa schopnosť určiť splniteľný plán pomocou skrátených algoritmov vetiev a hraníc výrazne zhoršuje. Preto pri používaní týchto algoritmov pre veľké projekty odporúčame nastaviť dlhšiu dobu vykonávania algoritmu. Naopak algoritmy využívajúce prioritné

pravidlá sa ukázali ako dostatočne efektívne. V tomto teste sme však neurčovali kvalitu navrhovaných rozvrhov, ale zisťovali sme len informáciu o tom, aké veľké projekty je daný algoritmus schopný vyriešiť za relatívne krátky čas. Na základe týchto výsledkov sme zistili, že nemôžeme vzájomne porovnávať všetky algoritmy, preto si ich rozdelíme do dvoch skupín. Prvú budú tvoriť skrátené algoritmy vetiev a hraníc a druhou budú prioritné pravidlá, pretože sme zistili, že jednotlivé algoritmy majú porovnateľné výsledky pri nájdení riešenia s použitím všetkých pravidiel. Najdôležitejšou informáciou v procese plánovania je však vykonať celý projekt za čo najkratšiu dobu. Preto vykonáme test, ktorý nám ukáže, aké optimálne sú navrhované rozvrhy, a za aký priemerný čas sme dokázali plán zhotoviť. Tento priemerný čas nájdenia najlepšieho riešenia si označíme ako $\bar{t}$. Na určenie samotnej efektivity plánu budeme používať jeho zistenú hornú hranicu *UB*, pričom budeme porovnávať najlepšiu nájdenú hornú hranicu UB_{min} pre daný projekt, určenú ľubovoľným algoritmom s aktuálne vypočítanou hornou hranicou UB_{real} projektu testovaným algoritmom. Výsledná hodnota ΔUB_{min} nám ukáže, aká je priemerná úspešnosť algoritmu zhotoviť optimálne riešenie. Najskôr vzájomne otestujeme tieto algoritmy: *aproximačná metóda*, *vyhľadávanie filtrovaním lúčov a VH metódy rozkladu*. Samozrejme nebudeme brať do úvahy projekty, ktorým nedokázal algoritmus zhotoviť splniteľný rozvrh, pretože opäť použijeme rovnané nastavenia týchto algoritmov, aké sme používali v predchádzajúcom teste, z ktorého vieme, že nie vždy sme dokázali nájsť riešenie.

Tabuľka 3: Porovnanie výsledkov skrátených algoritmov typu vetiev a hraníc

	Aproximačná m.	Vyhľadávanie filt. lúčov	VH m. rozkladu 1	VH m. rozkladu 2
$\bar{t}$	9,5s	11,6s	6,2s	3,4s
$\Delta UBmin$	89,67%	69,74%	83,21%	94,61%

Z tabuľky 3, obsahujúcej výsledky testu, môžeme prehlásiť, že skrátené algoritmy vetiev a hraníc sa ukázali ako efektívne v hľadaní najlepšieho možného riešenia, pričom priemerná doba určenia takéhoto plánu netrvala viac ako 12 sekúnd. Ako najviac efektívna sa ukázala metóda rozkladu vetiev a hraníc 2, ktorá nielenže pridá časové obmedzenia do cyklových štruktúr, ale naviac odstráni takzvané „spiatočné oblúky“. Naopak, najmenej efektívny bol pre testovanú množinu algoritmus vyhľadávania filtrovaním lúčov, čo však mohlo mať za následok pevne určené maximálne množstvo vrcholov stromu. Z toho vyplýva, že výsledky úzko súvisia s nastavením parametrov, ktoré sú využívané algoritmami, pretože pomocou nich môžeme riadiť presnosť a dôraz na jednotlivé obmedzenia. Následne si spravíme takýto test na porovnanie jednotlivých prioritných pravidiel, avšak naviac si určíme aj informáciu o tom, koľko percent projektov bolo pomocou pravidla vyriešených, aby sme vedeli vedeli určiť ich reliabilitu.

Z výsledkov testu zobrazených v tabuľke 4 je na prvý pohľad zrejmé, že tieto algoritmy dokážu za veľmi krátku dobu určiť rozvrh projektu, pričom v takmer každom prípade sa jedná o najlepší nájdený rozvrh. Keď sa však pozrieme na počet nájdených riešení podľa daného pravidla, ide o dosť veľké rozdiely. Vezmime teda do úvahy „relevantnosť“ rozvrhov a percento určenia rie-

Tabuľka 4: Porovnanie výsledkov prioritných pravidiel

	LPF	LST	MST	MTS	RSM
$\bar{t}$	8,17ms	10,12ms	10,83ms	13,77ms	3,18ms
$\Delta UBmin$	93,22%	96,98%	94,91%	96,77%	88,72%
Riešenia	57%	86,7%	75,09%	92,83%	30,03%

šenia projektu, pretože doba trvania výpočtu je v tomto prípade zanedbateľná. Potom môžeme prehlásiť, že pre našu testovanú množinu je najefektívnejšie používať prioritné pravidlo *MTS*, pretože pomocou neho sme dokázali určiť najviac riešení, pričom takmer všetky boli najlepšie, aké sme dokázali určiť. Pravidlo *RSM* sa neukázalo ako veľmi efektívne, čo mohlo byť spôsobené tým, že naše projekty využívali len päť obnoviteľných zdrojov.

9 Záver

Cieľom diplomovej práce bola implementácia niekoľkých algoritmov, ktoré by slúžili na nájdenie riešenia problémov označovaných ako RCPSP - resource constraint project scheduling problem. V úvodných kapitolách sme si predstavili základné pojmy spojené s touto problematikou, návrh dátových štruktúr používaných samotnými algoritmami a určili sme si základné koncepty potrebné pre nájdenie riešenia. V ďalších kapitolách sme sa venovali samotným algoritmom. Pri realizácii riešenia sme používali metódu vetiev a hraníc a jej skrátené verzie, a tiež metódy využívajúce prioritné pravidlá spracovávania aktivít. Popísali sme si jednotlivé kroky určenia splniteľného rozvrhu a všetky postupy potrebné pre ich nájdenie. Následne sme si popísali prácu s aplikáciou, jej vstupný súbor a samotnú interakciu s používateľom.Na základe výsledkov skrátených algoritmov vetiev a hraníc 3 a prioritných pravidiel 4, môžme prehlásiť, že implementované algoritmy sú vhodné na riešenie rozvrhovacích problémov s obmedzenými zdrojmi. Tieto algoritmy je možné aplikovať na takmer všetky rozhodovacie problémy používajúce obnoviteľné zdroje. Problém plánovania je stále aktuálny, a preto sa neustále vyvíjajú nové spôsoby jeho riešenia, napríklad s použitím genetických algoritmov. V mnohých prípadoch môže výsledný rozvrh ušetriť firmám mnoho času, a taktiež im môže pomôcť zefektívniť využívanie dostupných zdrojov. Všetky nami implementované algoritmy vychádzali z poznatkov, ktoré vo svojej publikácií uvádzajú Franck, Neumann, Schwindt (2001) [4].

Bc. Viliam Frolo

Literatúra

[1] S. H. Dorit - *Graph Algorithms and Network Flows*, Floyd-Warshall algorithm for all pairs shortest paths,pp. 41-42, Berkeley, University of California 2014.

[2] S. H. Tarjan - *DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*, SIAM J. Comput, 1(2), pp. 146-160, 1972.

[3] K. Neumann, Schwindt C., Zimmermann J. - *Project Scheduling with Time Windows and Scarce Resources*, Temporal and Resource-Constrained Project Scheduling with Regular and Nonregular Objective Functions, pp. 83-84, Springer-Verlag, 2002.

[4] B. Franck, K. Neumann, C. Schwindt - *Truncated branch-and-bound, schedule-construction, and schedule-improvement procedures for resource-constrained project scheduling*, OR Spektrum 23, pp. 297-324, Springer-Verlag, 2001.

A Vstupné súbory

Vstupné súbory, spĺňajúce požadovaný formát, sú dostupné na stránke `https://www.wiwi.tu-clausthal.de/en/abteilungen/produktion/forschung/schwerpunkte/project-generator/rcpspmax/` v kategórii Testsets UBO.

B Príloha na CD

Na priloženom CD sú dostupné nasledujúce zložky:

- Diplomová práca - obsahuje text diplomovej práce
- Zdrojové kódy - obsahuje projekt aplikácie vo Visual Studio 2015
- Aplikácia - obsahuje spustiteľnú verziu aplikácie a potrebné knižnice pre spustenie
- Dáta - obsahuje zadania projektov rozdelených podľa počtu aktivít